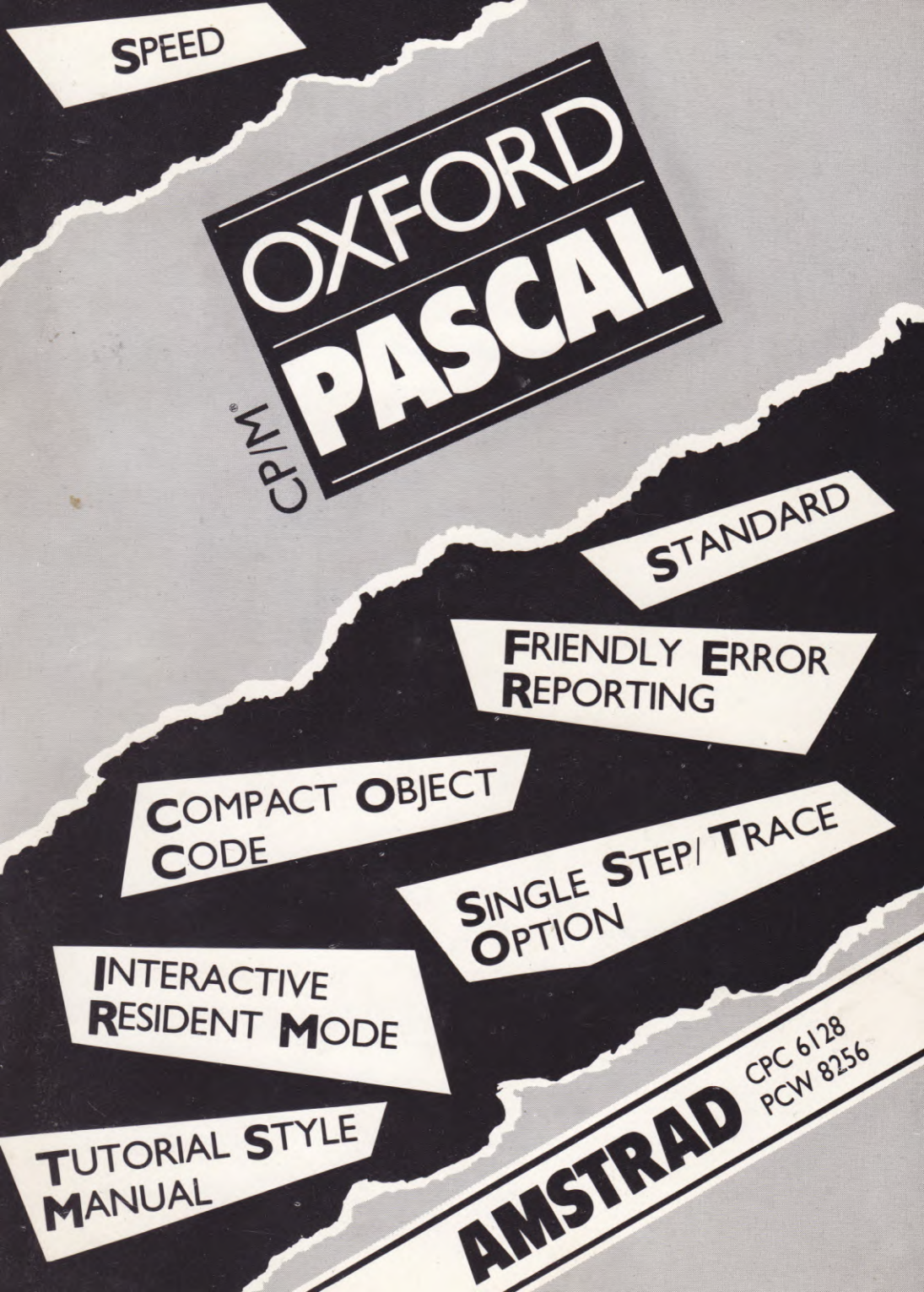
SPEED
OXFORD
CP/M®
PASCAL
STANDARD
FRIENDLY ERROR REPORTING
COMPACT OBJECT CODE
SINGLE STEP/ TRACE OPTION
INTERACTIVE RESIDENT MODE
TUTORIAL STYLE MANUAL
AMSTRAD
CPC 6128
PCW 8256

OXFORD PASCAL FOR CP/M

USERS' MANUAL

Copyright (c) 1979/80 All rights reserved.

Systems Software (Oxford) Ltd.
16B Worcester Place, Oxford OX1 2JW, England
Telephone (0865) 54195 Telex 83147 OXSOFT

CONTENTS

I. Introduction to OXFORD PASCAL

PASCAL is a powerful high level computer language written by Niklaus Wirth of Zurich, Switzerland.*
It can be efficiently implemented on small computers as well as large mainframes, offering numerous advantages over other popular microcomputer languages such as BASIC.
Some of these advantages are:

ALGOL-like block structure

Meaningful variable names

Powerful data structuring techniques

User-defined data types and constants

Excellent function and subroutine linkage

Recursive calls

Clean, modern flow of control

Runtime error checking

Dynamic variable allocation

Greater standardisation

High speed of execution

Greater program legibility

OXFORD PASCAL is an implementation of standard PASCAL designed specially for microcomputers. It offers all the features of this powerful language together with some useful enhancements for the personal computer user.

The Amstrad CP/M version for the CPC6128 and the PCW256 has two modes of operation. In the simplest mode the Pascal compiler co-resides in RAM with the user's program. This is ideal for learning the language or writing small programs which do not need the disk. Most Pascal commands are available in this mode except those involving diskette files. For more complex programs the disk-based compiler can be used to give the full power of the language.

* PASCAL USER MANUAL AND REPORT BY JENSEN AND WIRTH
  ---Springer-Verlag 1975

Some implementation information

MAXINT = 32767

type INTEGER = -32768...32767

type CHAR = the ASCII set

set values: must be in 0...127(ie set of char is allowed)

real numbers: accuracy: 6 1/2 digits
range: approx 1E-38 to 1E38

default output formats: integer : 7 characters
real : 12 characters
boolean : 6 characters
char : 1 character
string : size of string

program size and complexity: No restriction, apart from exceeding the total memory capacity of the system (STACK OVERFLOW is printed)

identifiers: first 8 characters must be unique

labels: first 8 digits must be unique

Extensions to standard Pascal

Dynamic specification of filenames
Input of strings
Hexadecimal numbers and hex I/O
Bit manipulation
Machine language interface
Memory , port and VDU screen access
Catchable I/O errors
Random number generator
Program chaining
CP/M directory maintenance
Separate compilation (linking)

II Beginner's Guide to Pascal

This section is a straightforward introduction to some of the features of Pascal.

PRODUCT DESCRIPTION

Your OXFORD PASCAL package contains the following items:

1) This Manual

2) A 3 inch disk containing the following files

PASCAL.COM	The Resident Pascal compiler
PAS.COM	The Disc Compiler
PASERROR.MSG	The error message file
RUN.COM	The Disc compiler's run time package
LINK.COM	The LINKER
LOCATE.COM	The LOCATOR
PASSYS.LIB	Library used by the LOCATOR
RECOVER.COM	Recovery program for Pascal

1. Getting Started - Write Statements

We assume that you have loaded CP/M as described in Amstrad's documentation. In order to run OXFORD PASCAL you must remove the CP/M disc and insert the Pascal disc. The screen should display the prompt

```
A>
```

To run Pascal, just type :

```
pascal
```

followed by a carriage return. You should get the PASCAL SIGN-ON MESSAGE:

```
OXFORD PASCAL
=============

(c) OCSS 1985.

XXXX bytes free.

E(dit), R(un), C(ompile), L(ist), T(race) ?
```

Let us start right away with a very simple programming example:

Example 1

First of all you must enter the program into the computer memory, and this is done using the EDITOR. So type the letter "e" (for edit), followed by the carriage return key (referred to in this manual as <return>). The computer should now respond with the prompt:

```
>
```

Which means that you are in edit mode.

The editor has many powerful commands, but to get things moving we will start with just one. Type:

```
a <return>
```

That is, the letter "a" followed by the return key. This tells the computer that you want to APPEND lines of program to the computer memory. Check each line carefully before finally entering it into the computer using the return key. If you make a mistake, you can erase the last key you typed by pressing the <DELETE> key. This works right up to the time you finally press <RETURN> for each line. Be especially carful about spelling and punctuation, and don't forget that full stop at the end.

Here is the program :

```
begin
write ('Hi there')
end.
```

When you have finished typing in the program, type a line containing just a full stop :

```
. <return>
```

This will signal to the editor that you have finished inserting text, and you should once again get the editor prompt:

```
>
```

Now type :

```
q <return>
```

to quit the editor. The computer should now print the Pascal prompt :

```
E(dit), R(un), C(ompile), L(ist), T(race) ?
```

All you have to do to run your program is to type

```
r<return>
```

If all goes well the computer should reply with something like the following:

```
Compiling
Program   0  0509
0 error(s)
Compilation complete,  5736 bytes free.
```

Do not worry about the details, but what is happening is that the computer is scanning your program and converting it into a numeric form which it can efficiently execute. (If you don't get the message: "0 error(s)", then you probably made a mistake in typing. You could try typing "k <return>" and starting again!).
Now the computer should automatically run your program, and print the message:

```
Hi there!
```

Once your program has been compiled, it can be run as many times as you like by typing:

```
r<return>
```

Each time the computer should print:

```
Hi there!
```

Now let us look at the program in more detail. The main body of a

Pascal program is always enclosed between the words BEGIN and END, the final END must be followed by a full stop. Pascal programs consist of a sequence of "statements" which are executed sequentially in the order they are written. Example 1 has one statement, a WRITE statement which tells the computer to write something on the screen, in this case the message "Hi there!" The object enclosed in the single quotes is called a STRING, and may contain any sequence of characters except <return>. Also, if a single quote is itself to be included in a string, it should be doubled up, so that the Pascal program

```
begin
write ('O''Brien''s string')
end.
```

would cause the message

```
O'Brien's string
```

to be printed on the screen.

Example 2

Other things can be printed besides strings. Try the following program. We will use the same steps as example 1 but we must remember to erase example 1 from the computer memory. So type:

```
k <return>
```

The computer will ask :

```
Sure ?
```

Reply with :

```
y <return>
```

to confirm that you wish to erase the program.

As before, type :

```
e <return>    (to enter the editor)
a <return>    (to append new program lines)
```

now type example 2 into the computer:

```
begin
write (3 + 4) ;
write (6 - 2 - 1)
end.
```

followed by

```
. <return>    (to stop appending)
q <return>    (to quit the editor)
r <return>    (to compile and run the program)
```

When the program is run the computer should print

```
7  3
```

Example 2 contains two statements, which must be separated by a semicolon. It has examples of INTEGER (whole number) arithmetic.

Now try the next example:

Example 3 multiplication and division

```
begin
write (6 * 7, 18 div 4, 18 mod 4, -(4 + 2) * 3)
end.
```

The computer should print

```
42   4    2  -18
```

In Pascal "*" means multiplication, DIV means integer division (ie with rounding towards zero), and "18 MOD 4" gives the remainder when 18 is divided by 4.

Note how brackets have been used to change the order of evaluating -4 + 6, or 2. This is because the computer does multiplications and divisions before it does additions and subtractions.

Any number of items can be printed using a single WRITE statement, provided that they are separated by commas.

Example 4 functions

```
begin
write (sqr (4 + 5), abs (- 44), abs (44), odd (3))
end.
```

The computer should print

```
81   44   44   TRUE
```

SQR, ABS and ODD are called "functions". There are many different

functions in Pascal.

SQR, followed by a number in brackets, gives the square of the number.

ABS gives the absolute value of the number.

ODD (3) is TRUE because 3 is odd.

The last function, ODD, gives a Boolean, or logical result, that is it can either be TRUE or FALSE. Boolean values are used a lot in Pascal so let us look at them more closely.

Example 5 Boolean expressions

```
begin
writeln  (true, false, 3 = 3, 3 = 4);
write    (3<>4,  5<6, 9 >=10);
end.
```

Should print:

```
TRUE FALSE TRUE FALSE

TRUE TRUE FALSE
```

because: 3 is equal to itself
3 is not equal to 4
etc.

```
=  means  "equal to"
<  means  "less than"
>  means  "greater than"
>= means  "greater than or equal to"
<= means  "less than or equal to"
<> means  "not equal to"
```

WRITELN is like WRITE but also generates a new line after printing all the values in brackets.

Example 6 Boolean expressions

These can get a bit complicated, but the computer evaluates them using the rules of logic.

```
begin
write  ((3 = 3) and (3<5),(3 = 4) or (3>11));
write (not true, not false, not (1 = 2));
end.
```

Gives the result:

```
TRUE FALSE FALSE TRUE TRUE
```

because both (3 = 3) and(3<5) are true
neither (3 = 4) nor (3>11) are true
(1 = 2) is false so not (1 = 2) is true

"x and y" is TRUE if both x and y are TRUE
"x or y" is TRUE if either x or y (or both) are TRUE
"not x" is TRUE if x is FALSE, and FALSE if x is TRUE.

Ch 2. PASCAL STATEMENTS

First a word about symbols. These are the building blocks of Pascal programs, and there are three main kinds:

1. Pascal keywords, such as BEGIN and END, which are reserved and can't be altered by the user. A complete list of these is given in the reference manual, section 1.1.

2. Special symbols such as . ; := .. <> etc.

3. Identifiers, which are names chosen by the user. They can be any sequence of letters or digits, but must start with a letter. For example:

```
i
Henrythe8th
PI
```

WARNING Identifiers are unique only if they differ in the first 8 characters, so that Henrythe7th and Henrythe8th are the same identifier in OXFORD PASCAL (and many other implementations).

Upper case letters are equivalent to their lower case counterparts so that PI, pi and Pi are all synonymous.

Some standard identifiers such as WRITE and WRITELN are predeclared in every version of Pascal.

These can be redefined by the user, however (in contrast to Pascal keywords).

IMPORTANT Pascal symbols can't contain imbedded blanks. "Henry the 8th" is not the same as "Henrythe8th", and "30 000" is not equivalent to the number 30000. ("30,000" would also be illegal). Note especially that ": =" cannot be used instead of ":=".

This aside, spaces, tabs and new lines may occur anywhere in a Pascal program, and are ignored.

Now we return to some actual examples of Pascal programs. Indentation is used by putting spaces in front of certain lines. This is optional, but helps to make the program clearer to humans.

Example 7 Variables and assignment.

```
var x,y :integer;
begin
  x:=3; y:=27;
  writeln (x,y);
  x:=4;
  y:=x+2;
  write (x,y, x+y);
end.
```

Should print:

```
 3   27
 4    6   10
```

The VAR declaration comes before the BEGIN, and informs the compiler that the identifiers x and y are "variables" which can take integer values. As the name implies, variables can change in value throughout the execution of the program. In line 3, the value of x is set to 3 and the value of y is set to 27. Then later, x is set to 4 and y is set to x + 2, or 4 + 2 = 6. Notice that y:=y+2 could also have been written setting y to 27 + 2 = 29. Variables can also be declared as BOOLEAN and many other types besides INTEGER.

Example 8 repetition using "FOR" loops.

```
var i : integer;
begin
  writeln ('going up');
  for i := 1 to 5 do writeln (i);
  writeln ('going down');
  for i := 5 downto -1 do writeln (i);
end.
```

Should print:

```
going up
1
2
3
4
5

going down

5
4
3
2
1
0
-1
```

The statement following the "FOR .. DO" (in this case a WRITELN statement) is repeated once with each value of the variable i.

Example 9 "if" statements

```
var i : integer;
begin
  for i:= 1 to 11 do
   begin
    write (i);
    if odd (i) then writeln (' is odd')
    else writeln (' is even');
   end
end.
```

The result should be:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

```
 6 is even
 7 is odd
 8 is even
 9 is odd
10 is even
11 is odd
```

'if" statements give the computer a choice of two statements to do, depending on the value of the Boolean expression. (Remember, Boolean expressions can either be TRUE or FALSE). The "else" part of a conditional statement is optional but --IMPORTANT-- "else" is never preceded by a semi-colon.

The WRITE and IF statements in our example are enclosed in BEGIN...END to make them act as a single statement to be repeated by the FOR loop.

Example 10 finding the average

This example introduces keyboard input, and a more general sort of loop.

```
var total, count, x : integer;
begin
  total:=0; count:=0;
  write ('Type some numbers: ');
  repeat
    read (x);
    total:=total+x;
    if x>0 then count:=count+1;
  until x=0;
  writeln ('The average is' total/count);
end.
```

When you run this program, the computer should invite you to type a series of numbers. Try typing:

```
3  47  5  199  0  <return>
```

The computer should reply with

```
The average is 6.35000E+01
```

The statement read (x) tells the computer to accept an integer from the keyboard and place its value in the variable x. If you type something the computer doesn't recognise as an integer, you might get the message

```
INTEGER READ ERROR  line 6
```

and the program will terminate.

The "/" operator gives division with a floating point, or REAL result (whereas DIV gives an integer result). More about REAL arithmetic later.

The "real" number was printed in "scientific" notation, which will be familiar to many calculator users. The number following the "E" represents a power of 10, so that 6.35 E+01 means "6.35 (times 10 to the power of +1) or 63.5.

The printing format can be changed to make it more legible by specifying the total number of characters you would like printed and the number of digits after the decimal point. (Rounding is done automatically). Thus:

```
Writeln ('The average is', total/count : 10 : 3)
```

would have printed

```
The average is     63.500
```

The number is printed with 4 leading blanks to give the 10-character field you specified.

The "repeat"..."until" loop simply executes the enclosed statements until the condition at the end turns out to be TRUE. In our example the loop is terminated when a zero is read from the keyboard.

Example 11 Case statement

This example introduces a slightly more elaborate way of choosing one of several statements:

```
var verse, i : integer;
begin
  for verse:=1 to 4 do
    begin
      writeln;
      for i := verse downto 0 do
       case i of
        3: writeln('three men');
        2: writeln('two men');
        1: writeln('one man');
        0: writeln('and his dog')
       end
    end
end.
```

This should result in the printout:

```
one man
and his dog

two men
one man
and his dog

three men
two men
one man
and his dog
CASE ERROR line 7
```

The error message was caused because in the last verse i becomes 4 and there is no corresponding label in the CASE statement.
Case labels can also be combined, for example

```
4,5,6 : writeln ('Many men');
```

A note on error messages

The "CASE ERROR" message is called a "runtime" error message because it occurs while the program is actually running. There are several such messages which you may encounter (see section 8).

By now you may have started to experiment with your own programs. (This is probably the best way of finding out what is and is not possible in Pascal). If so, you will sooner or later get one of the compiler's error messages. This may also happen if you make a mistake in typing one of the examples. The simple program

```
var x : boolean;
    x : integer;
begin
  read (x);
  write (x)
end.
```

Would cause the following message during compilation:

```
compiling
---ERROR type 46 line 2 near X
...IDENTIFIER DECLARED TWICE
program  0    050d
1 error(s)
Compilation complete.
```

The error number given was 46. There are over 100 possible errors, and so an appropriate message is selected by the compiler from a disk file (PASERROR.MSG), in this case: "Identifier declared twice". Referring to the second line of the program, that's exactly what we

have done.

NOTE---the line number may sometimes be out by 1 line or even more, depending on how long the compiler needed to detect the error.

Typing "L" in response to the prompt:

```
E(dit), R(un), C(ompile), L(ist), T(race) ?
```

gives a listing of the program as well as compiling it. All the line numbers and errors are marked. In our example:

```
1 var x :boolean;
2     x <- ERROR 46
..IDENTIFIER DECLARED TWICE
2     x : integer;
3     begin
4       read (x);
5       write (x)
6     end.
```

The full version of line 2 is retyped underneath the error report.

The computer will not let you run a program if there are any compiler errors.

Correcting errors

This can be done using the editor, without having to retype the whole program. To correct the small example above you might type:

```
e <return>
```

to get into the editor. Then

```
2d <return>
```

to delete line 2. Now we can look at the whole program by typing

```
1,$p <return>
```

This will list lines 1 through $ ($ means the last line in the program):

```
var  x : boolean
begin
  read (x);
  write (x)
end.
```

Line 1 is still wrong. We want to read and write integer, so we type

```
l <return>
```

to go and display line 1. Then the command

```
s/boolean/integer/ <return>
```

makes the substitution and retypes the line.

1,$p <return> should now print the correct version of the program.

```
var  x : integer;
begin
  read (x);
  write (x)
end.
```

Now to quit the editor and run the program all we need to do is type:

```
q <return>
r <return>
```

The program doesn't do much, just reads a number from the keyboard and prints it out again.

For a more complete explanation of how the editor works, see the separate editor manual. Also see the command summary (section 7), which explains how to load and save your Pascal program on diskette.

WHILE statement

There is another sort of loop in Pascal, besides REPEAT and FOR loops.

The WHILE statement is like the REPEAT statement except that the test is done at the beginning of the loop (so that the loop need not be executed at all). Also, like the FOR loop only one statement may be repeated (or a sequence of statements enclosed in BEGIN and END).

Example:

```
i:=1;
while i <= 5 do
  begin
    writeln (i);
    i:=i+1;
  end;
```

Has the same effect as

```
for i:=1 to 5 do writeln (i);
```

3. More about data types in Pascal

Example 12 - Floating point numbers.

```
begin
  writeln (3.3, 33.0, 330.0, 0.33);
  writeln (-3.3E3, 3.3E-1, 4.5+2.1)
end.
```

The computer should print

```
 3.30000E+00  3.30000E+01  3.30000E+02  3.30000E-01
-3.30000E+03  3.30000E-01  6.60000E+00
```

The presence of either a decimal point or an exponent (the "E" part) in a number tells Pascal to treat it as a floating point or a REAL number.

```
3.3E3 means 3.3 times (10 to the power of 3)
in other words 3.3 x (10x10x10) or 3300
```

Floating point numbers in Pascal have an accuracy of 6 1/2 digits and may range in size from about 1E-38 to 1E38. In contrast to integer, you should not expect Pascal real arithmetic to be exact. This means, for example, that 4.0 may in fact be printed as 3.999999. Also, you can't rely on testing real numbers for equality. 2.0 + 2.0 = 4.0 may not always be true!

Example 13 floating point arithmetic

```
var x, y: real;
begin
    x := 9.1;
    y := 8.7;
    writeln (x+y: 7:2, x-y: 7:2, x*y: 7:2, x/y: 7:2);
    writeln (sqr (x) : 7:2, sqrt (x): 7:2, abs (x): 7:2);
    write (trunc (x), trunc (y), round (x), round (y));
end.
```

Should print:

```
17.80  0.40  79.17  1.05
82.81  3.02   9.10
    9     8     9     9
```

We have already met +, -, * and /. They are used to mean addition, subtraction, multipliation and floating point division (DIV means

integer division. DIV and MOD shouldn't be used with reals).

SQR (X) means the square of X
SQRT (X) means the square root of X
ABS (X) gives the absolute value of X
TRUNC (X) gives the integer (whole number) part of X
ROUND (X) rounds X to the nearest integer.

Some other useful mathematical functions are

SIN (X) gives the sine of X (X is in radians)
COS (X) gives the cosine of X (X is in radians)
ARCTAN (X) gives the angle whose tangent is X (in radians)
LN (X) gives the natural logarithm (base e) of X (for X > 0)
EXP (X) gives the number e raised to the xth power.

1 radian = 57.29578 degrees

e= 2.718281

Example 14 Output formatting, constants.

```
program waves;
const f1 = 0.5; f2 = 0.05; amplitude = 19;
var x1, x2, y :real;
begin
    x1:=0;  x2:=0;
    repeat
    x1:=x1 + f1;
    x2:=x2 + f2;
    y:=sin (x1) * sin (x2) * amplitude;
    writeln ('x': round (y) + amplitude);
    until false;
end.
```

The program should print an amplitude - modulated sine wave.

Because of the REPEAT..UNTIL FALSE loop, example 14 will continue printing almost forever (at least until x1 or x2 becomes too large!) One way of stopping it would be to turn off the power, but if you did that you would lose the program. A better way is simply to press the ESCAPE key.
The computer should print:

BREAK AT LINE xxxxxx

Where xxxxxx is the line it happened to be executing when you pressed ESCAPE. (If this doesn't happen, try again)

The "Program" header is optional in OXFORD PASCAL and in this case simply serves to give the program a name: WAVES. The name has no significance to the computer; it's merely there as an aid to documentation.

Any text enclosed between the pairs of symbols (* and *) is also ignored by the compiler. This facility can be used to write comments which help human readers to understand the program. Constants, introduced by the keyword CONST, are values which don't change throughout the program. It is an error to use a constant on the left of an assignment statement or as a parameter in the READ statement.

"CONST" declarations are useful for giving names to special values (for example PI = 3.1415926), and they make the program easier to change later. Try using the editor to alter the frequencies f1 and f2 and the amplitude to give different wvae patterns in example 14.

Note how the program uses a field width specification (a colon followed by an integer value) to tell the computer how many characters to allocate to the 'x' when printing. If too many characters are asked for, enough spaces are printed to make up the difference. If not enough are asked for, the string is truncated on the right, for example

```
write ('Hi there' :5)
```

would print:

```
Hi th
```

Numeric values, however, are always printed in full even if too few characters are specified.

Example 15 Graphics

```
var line, i: integer;
begin
    page;
    for line:=1 to 16 do
    for i:=1 to 64 do
    if odd (i) then write (chr(143))
    else write(chr(143))
end.
```

This should fill the screen with a pattern. CHARACTERS in Pascal are strings of length 1, for example:

```
'x'  '?'  ''''
```

They belong to the data type "Char", which has 128 possible values in OXFORD PASCAL, corresponding to the ASCII character set.

The function ord (ch) gives the ASCII integer code (between 0 and 127) for the character ch, while chr (x) gives the character represented by the integer x. So ord ('?')=63 and correspondingly chr(63)='?'.

NOTE - the OXFORD PASCAL data type "char" has been extended to the range 0..255 to allow the graphics font on certain computers to be used. Two such characters were used in the program above. Try writing programs to give different patterns, using the available characters on your machine.

The statement PAGE simply clears the vdu screen.

Example 16 Arrays

Suppose you wanted to read in some numbers and print them out in reverse order. You would have to store the numbers somewhere because you can't start printing until the last number has been read. If you knew that there were always going to be three values, you could write:

```
var x1, x2, x3 :integer;
begin
    write ('Type 3 numbers : ');
    read (x1, x2, x3);
    writeln (x3);
    writeln (x2);
    writeln (x1);
end.
```

But for 50 values this would get a bit tedious!

The answer is to use an array variable:

```
const n=3;
var x : array [1..n] of integer;
    i : integer;
begin
    write ('Type ', n:1, ' numbers: ');
    for i:=1 to n do read (x[i]);
    for i:=n downto 1 do writeln (x[i]);
end.
```

Running the program and typing the data:

```
463  79  980
```

Should give the result:

```
980
 79
463
```

The declaration of x really declares n variables which can be referred to by giving an index in square brackets. The elements of the array x are thus x[1], x[2], ..., x[n].
The constant n was used so that the number of values read in by the program can easily be changed by altering just one line.

Array elements can be any valid Pascal data type, including another array. This allows two dimensional (or indeed any dimensional) arrays, and a chessboard for example may be represented as:

```
var chessboard: array [1..8] of array [1..8] of chesspiece;
```

Where chesspiece is some suitable data type, probably a user defined type (more about this later). The 5th square of the 3rd row of the chessboard could then be referred to as:

```
Chessboard [3] [5]
```

Because arrays of arrays are used often in Pascal programs, the abbreviation "chessboard [3,5] " is allowed, and similarly in the declaration:

```
var chessboard : array [1..8, 1..8] of chesspiece;
```

This can be extended to arrays of any dimension.

Defining your own data type

None of the data types so far mentioned (integer, real, boolean, or even char) would be really suitable for describing a piece on a chessboard, so Pascal lets you define your own. This may be done in a TYPE DECLARATION, for example:

```
type chesspiece = (pawn, knight, bishop, rook, queen, king);
```

Then a variable of type CHESSPIECE could take any of these six values, for example:

```
var mypiece, yourpiece:chesspiece
begin
    :
    :
    mypiece:=rook;
    yourpiece:=queen;
    :
```

Type declarations come after constant declarations and before variable declarations. The identifiers used in an 'enumerated' data type like CHESSPIECE must be unique, they can't appear in other enumerated types or be declared as constants or variables. Enumerated types are ordered so that our chess pieces can be compared using =, > etc:

```
king>queen
queen>rook
```

and so on:

```
    :
```

Three functions are also defined: PRED, SUCC and ORD

```
 pred (x) gives the value preceeding x
 succ (x) gives the value succeeding x
 ord (x) gives the position of x within the data type.
                  (starting with pawn = 0)
 so pred (bishop) = knight
    succ (rook) = queen
but pred (pawn) and succ(king) are both meaningless
    ord (knight) = 1
    ord (rook) = 3, and so on.
```

Example 17 The sieve of Eratosthenes.

This program finds and prints all the prime numbers between 2 and 127.

```
program Eratosthenes;
const n=127;
var sieve : set of 2..n;
    number, i : integer;
begin
    sieve := [2..n];
    for number := 2 to n do if number in sieve then
     begin
      writeln (number);
      for i := 2 to n div number do
                  sieve := sieve - [i*number] ;
     end;
end.
```

A prime number is divisible only by itself and 1. Our "sieve" used for finding the prime numbers, is a new type of variable called a SET variable.

Sets in Pascal are collections of objects enclosed in square brackets. Either an object is in a set or it is not, so

```
    [1,2,3]
    [2,3,1]
and [1,1,3,3,2] are all equivalent.
```

The abreviation x..y in a set means all the items between x and y inclusive, so

```
    [1..4, 10] = [1,2,3,4, 10]
```

We can test whether an item is in a set by using the operator IN. Thus "4 in [1..5]" will give the Boolean result: TRUE.

The type of a set can be any scalar type (ie not an array or a set) except REAL. Values are restricted to the range 0..127 (so "set of char" is acceptable).

Now back to our sieve program. Starting with the number 2 and working upwards, if a number is still in the sieve then it's a prime. We simply eliminate all multiples of that number from the sieve because they are not prime. Operations allowed on two sets x and y are:

x + y which gives the set of all items present in either x or y or both.

x - y which gives all items in x which are not also in y.

x * y gives all items present in x and also present in y.

x = y tests if two sets are equal

x<>y tests if two sets are not equal.

x<=y tests if all items in x are also in y.

x>=y tests if all items in y are also in x.

4. Procedures and functions

Example 18 procedures

```
var ch: char;
procedure lineof (wotsit :char);
    var i: integer;
    begin
      for i:=1 to 30 do write (wotsit);
      writeln;
    end; (* of procedure "lineof" *)
begin (* of main program *)
    lineof ('?');
    writeln;
    for ch:= 'a' to 'f' do lineof (ch);
end.
```

you don't need to type the comments (* ...*) if you don't want to. These are there to help explain the program.

The computer should print:

```
??????  ..........
aaaaaa  ..........
bbbbbb  ..........
cccccc  ..........
dddddd  ..........
eeeeee  ..........
ffffff  ..........
```

Procedures are used to separate sections of code from the main program, either to make what the program does clearer by dividing it up functionally, or to allow the same code to be "called" from various parts of the program. The procedure "lineof" has a PARAMETER "wotsit" (which takes the data type CHAR). When lineof is called it must be followed by a corresponding actual parameter in brackets. Then "lineof" simply writes a line of wotsit's on the screen.

If a procedure has no parameter then the brackets are omitted. The variable I is "local" to the procedure lineof, the main program doesn't know about it. However lineof could if necessary access the 'global' variable CH. Using local variables helps conserve storage, since they are destroyed when the procedure finishes. Procedures are really mini-programs in their own right. They can have their own constant and data type declarations and even their own procedures.

"WOTSIT" is called a VALUE parameter because a value is substituted for it when the procedure is called. Lineof could change the value of wotsit without affecting the main program. VARIABLE parameters on the other hand are substituted with variables when the procedure is called.

Example 19 Variable parameters

```
var x,y :integer;
procedure swap (var a,b : integer);
    var temp: integer;
    begin
      temp:=a;
      a:=b;
      b:=temp;
    end;
begin
      x:=4;  y:=77;
      writeln (x,y);
      swap (x,y);
      writeln (x,y);
end.
```

This should give the result

```
 4    77
77    4
```

Note that it is alright to have local variables, constants and parameters with the same names used in the main program. For example

```
procedure swap (var x,y : integer);
```

The computer won't get confused (but you might!). The variable parameters a and b are used by SWAP as a means of returning a result to the main program. Another way of returning a value is to define a function.

Example 20 Defining a function

```
var i : integer;
function cube (x : integer) : integer;
    begin
        cube := x*x*x;
    end;
begin
    for i:= 1 to 20 do writeln ('The cube of ',
                    i : 2,' is', cube (i));
end.
```

The program should print some numbers and their cubes. Apart from having to specify a return value, functions are just like procedures.

Example 21 Recursion

A recursive function or procedure is one that calls itself. Using recursion can give neat solutions to mind-bending problems like the "Towers of Hanoi". In this well known puzzle, there are three piles of discs. To start with piles 2 and 3 are empty, and the first pile has a number of discs stacked in order of size, smallest at the top. The game is to get all the discs in the same order (smallest on top) over to the 3rd pile, moving only one at a time, with no disc ever resting on a smaller disc.

```
program Hanoi;
var ndiscs: integer;
procedure move (source, destn, spare: 1..3; n:integer);
begin
    if n>1 then move (source, spare, destn, n-1);
   writeln('Moving from', source : 2, ' to ', destn : 2);
    if n>1 then move (spare, destn, source, n-1);
end;
begin
    write ('How many discs ? ');
    read (ndiscs);
    writeln;
    move (1,3,2,ndiscs);
end.
```

Moving one disc is trivial. To move n discs we first move the top (n-1) to the spare pile and then move the bottom one. Then the top (n-1) are moved using the same technique.

Recursive programs are not always the most efficient, though. They tend to gobble up memory because the computer has to save the variables for each call on the stack. If you make ndiscs too large the computer will run out of memory and print STACK OVERFLOW -- line xxxx. The same will happen if you declare more variables in a program then you have memory available, or if you try to compile too large a program.

Text Files

Text files are special Pascal variables having the data type TEXT which are essentially streams of characters with no fixed size. Three are preassigned in OXFORD PASCAL. "Input" and "output" are associated normally with the keyboard and the VDU display respectively. "Printer", which is not standard Pascal, corresponds to the CP/M list device.

By default, "Input" is implied in READ, READLN, EOLN and EOF and "Output" is implied in WRITE, WRITELN and PAGE. So for example,

```
EOF is really short for EOF (INPUT)
WRITELN ('Hi!') is really short for WRITELN (OUTPUT, 'Hi!')
```

Each textfile has an associated buffer variable of type CHAR (the file name followed by an upward arrow), for example: input^.

The procedure call:

get (input) reads the next character from the keyboard and puts it in the variable input ^.

put (output) writes the contents of output ^ to the VDU. So if X is a character variable,

read (x) is equivalent to x :=input^; get (input)
write (x) is equivalent to output^ :=x; put (output)

Newlines are special characters in textfiles. When a file buffer contains one, assignments like

```
x := input^
```

will set x to a space. Also, the end of line function EOLN will return TRUE.

READLN is like READ but afterwards skips to the beginning of the next line by doing:

```
while not eoln do get (input);
get (input);
```

This is awkward for interactive programs because the next line of input must be typed before the program can proceed, since input ^ is supposed to contain the first character of the next line. It's better to use READ and skip any leading spaces before you read the next value. (This is done when reading numeric values anyway).

Initially when the program is run, input ^ contains the newline you typed at the end of the "RUN" command.

Example 22 strings

```
program revwords;
const linesize = 64;
type string=packed array [1..LINESIZE] of char;
var word:string;
    nchars, i :integer;
procedure skipblanks;
    begin while (input^ = ' ') and not eoln do
        get (input) end;

procedure swap (var s:string;i,j:integer);
var temp:char;
begin
    temp:=S[i];
    S[i]:=S[j];
    S[j]:=temp;
  end;

begin
  repeat if eoln then readln;
    skipblanks;
    while not eoln do
      begin
        nchars:=0;
```

```
            repeat
              nchars:=nchars+1;
              read (word [nchars]);
            until input^ = ' ';
            for i:=1 to nchars div 2 do
              swap (word,i,nchars -i+1);
            write (word:nchars, ' ');
          end;
        writeln;
        writeln;
    until false
  end.
```

The program reads words and writes them out with the letters reversed. For example

```
  Mary had a little lamb <return>
```

would print

```
  yraM dah a elttil bmal
```

To stop the program, hit <escape>.

Strings of size n in Pascal are treated as "packed array [1..n] of char". Packed arrays are like ordinary arrays but are compressed to optimize storage. Packed array ELEMENTS can't be used directly as VAR - parameters (but whole arrays can, as in the example).
Examples of string operations in Pascal:

```
  var  str:  packed array [1..4] of char;
  begin
       str:='when';
       writeln (str>'what')
  end.
```

Would print TRUE because "when" is greater than "what" (Lexographically, i.e. dictionary order).

5. Advanced features

Example 23 Records

```
    Program clock;
    const delay = 500; (* approx. for Triton *)
    var i:integer;
        clock: record
                 hours: 0..23;
                 minutes, seconds :0..59;
        end;
    begin
        write ('Enter the time, in hours minutes seconds : ');
```

```
        read (clock. hours, clock. minutes, clock. seconds);
        with clock do repeat
             for i:= 1 to delay do (*nothing*);
             seconds:=(seconds + 1) mod 60;
             if seconds=0 then
              begin
                minutes:=(minutes+1)mod 60;
                if minutes=0 then
                  hours:=(hours+1)mod 24;
              end;
             writeln (hours:1,':',minutes:1,':',seconds:1);
        until false;
    end.
```

The program should print out the time roughly every second, for example:

```
    Enter the time, in hours minutes seconds : 1 39 56
```

Should print:

```
    1 : 39 : 57
    1 : 39 : 58
    1 : 39 : 59
    1 : 40 :  0
        etc
```

Records are a way of combining several conceptually related variables into one structure. The record can then be treated as a whole or the parts can be accessed individually using the dot notion.

The WITH..DO statement tells the computer to treat the elements of "clock" as though they were locally defined individual variables for that statement, removing the need for the"clock." prefix.

Record elements can be any type (for example other records or arrays), and in addition an optional "variant" part is allowed (see reference manual).

Example 24 Pointers

```
var p,q : ^ integer;
begin
    new (p); new (q);
    p^ :=3;
    q^ :=4;
    write (p^,q^);
end.
```

Should print

```
3   4
```

The variables p and q are not integers but 'pointers' to integer variables. The actual space for the variables to be stored at is created "dynamically" (in other words while the program is running) by the procedure NEW. This allows programs to create variables as required. A major use of pointers is in processing linked lists:

Example 25 Reversing a line of characters using a list

```
program revchars;
type itempointers=^item;
     item = record
              value:char;
              next: itempointer
     end;
var list,p: itempointer;
begin
    list:=nil;
    repeat
        new (p);
        read (p^.value);
        p^.next:=list;
        list:=p;
    until eoln;
    repeat
        write (p^.value);
        p:=p^.next;
    until p=nil;
end.
```

The input: Mary had a little lamb.

Should give the result: bmal elttil a dah yraM

This program defines a record containing a pointer to itself. (A recursive definition). Linked lists give very flexible storage but you have to keep careful track of what points to what.

In standard Pascal the procedure DISPOSE (P) releases the storage assigned to the pointer p^ and can be used when p^ is no longer needed.

In OXFORD RESIDENT PASCAL, dispose has no effect. (However, it is usually possible for programs themselves to implement some sort of "free list" of unwanted items). The Pascal keyword "nil" is a pointer value which points to no variable.

Example 26 "goto" statements

```
label 294, 33;
begin
     33: writeln ('This should be printed');
         goto 294;
        writeln ('This shouldn''t');
    294: writeln ('Stuck in a loop');
         goto 33
end.
```

This should print:

```
This should be printed
Stuck in a loop
This should be printed
Stuck in a loop
etc.
```

"labels" used with goto statements must be integers and should be declared before constants, data types and variables. GOTO'S should be avoided where possible because they destroy the structure of the program. A common use, however, is for "disaster" exits from nested procedures or statements. Jumping INTO a loop or a procedure will cause unpredictable results.

Extensions to standard Pascal

These are described in the reference manual, sections 10. One useful procedure is VDU:

Example 27 "poking" the vdu screen

```
var i: integer;
begin page;
    for i:=1 to 40 do vdu (i mod 4,i,'x');
end.
```

This should produce a pattern of x's on the screen. Vdu (i,j,ch) stores the character ch at row i, column j.
Remember, PAGE clears the VDU screen

6. Disk based operation

So far this manual has been concerned only with using the resident compiler, which is always in RAM. While this may provide an ideal environment for learning Pascal, it necessarily restricts the number of commands available, and the space remaining for user programs.

As you become familiar with Pascal, you will probably want to write larger programs. Using the disk-based compiler and linker, Pascal programs of thousands of lines may be run, and this may be extended even further by using program chaining.

Although your Amstrad computer has only a single disc drive it is possible to simulate the existance of further drives by swoping discs. This feature is described more fully in Amstrad's documentation however, in general it is possible to issue commands which assume a second drive. Whenever such a command is issued the operating system will ask you to change discs. In the remainder of this manual therefore, we will refer to drives A and B as if these actually existed. Whenever a change of disc is required, the operating system will prompt you to do so.

The disk-based compiler operates directly on CP/M disk files and provides a superset of the facilities of "resident Pascal", including a full set of disk commands.

Pascal source files may be created using the CP/M editor "ED", or you can use programs edited in resident mode and saved on disk. The source file name should end in the extension ".PAS", for example:

```
MYPROG.PAS
```

Then, in response to the CP/M prompt :

```
A>
```

the command:

```
pas myprog
```

will compile the textfile MYPROG.PAS and produce a relocatable object file MYPROG.OBJ, as well as a listing file MYPROG.PRN which includes line numbers and error messages.
The compiler output should be something like :

```
Pascal compiler v x.x
(c) Copyright OCSS 1985

program   0   0009
0 error(s)
Compilation complete.
```

The object file can be loaded and run simply by typing:

```
run myprog
```

Note that we have assumed that the files PAS.COM, RUN.COM are present. In addition the file PASERROR.MSG must be on the CURRENTLY LOGGED DISK (in this case disk A:).

The name of each procedure or function is printed out as it is compiled, together with its static nesting level (0 for the main program, 1 for outer level functions and procedures, and so on). A hexadecimal address is also printed, giving a rough idea of its relative position in memory

An extended form of the compile command is:

```
pas myprog.abc
```

where : "a" is a letter denoting the disk drive on which the souce file resides (A, B etc). "b" is the disk drive to which the object file is to be sent (Z = no object file generated). "c" is the disk drive to which the listing is to be sent (X = send to console, Z = no listing file generated).

This allows large programs to be compiled more quickly by skipping the listing, for example :

```
pas myprog.aaz
```

A compact compilation, without any line numbers or variable range checks generated in the object code, may be specified by typing a "-" before the carraige return, for example:

```
pas myprog -
```

or

```
pas myprog.abz-
```

The following table summarises the differences between resident and disk mode:

Resident Mode	Disk Mode
Compiler always in RAM	Compiler only in RAM during a compilation
Built-in Editor	Separate Editor
Pascal source and object code in RAM	Source and object code held in disk files
Language differences (see reference section for details):	
Console and printer only	All file types supported
	Disk files fully supported
	PACK, UNPACK implemented
DISPOSE is a no-op	DISPOSE fully implemented
	Program chaining allowed

Linker

For large programs it is desirable, (and when compiling on small systems often physically necessary) to have some form of modularization. Several Pascal source files with inter dependent functions and procedures may be compiled separately and their object files later "linked" into one file. The "locate" command may also be used to produce directly executable CP/M command files.

Examples:

```
link prog=myprog yourprog anyprog
```

links the files MYPROG.OBJ, YOURPROG.OBJ and ANYPROG.OBJ

into one object file PROG.OBJ

Restrictions

(a) The programs being linked must have identical variable declarations at the outer program level.

(b) Each outer-level function or procedure may only be defined in one file.

(c) If the other files need to refer to this function or procedure, a duplicate header should be included, with the body replaced by the keyword "extern".

(d) The first file in the list is assumed to contain the main program. (The other files would normally just contain a dummy main program:

```
begin
end.
```

Linker example:

file f1.pas:

```
program test (input, output);
var i: integer;
procedure x; extern; (* x is defined in the other file *)
procedure y;
  begin
    write (i)
  end;
begin     (* main program *)
  x
end.
```

file f2.pas:

```
program testpart2(input, output);
var i : integer; (* var's must be identical to f1 *)
procedure y; extern; (* y is defined in f1 *)
procedure x;
  begin
    i:=3;
    write ('three =');y;
  end;
begin
end.
```

The command sequence might be

```
pas f1
pas f2
link test=f1,f2
run test
```

The program should print "three = 3"

Including other files in a compilation

If the character "£", followed immediately by a diskette file name, is placed at the beginning of a pascal source line, then this indicates to the compiler that the contents of the specified file are to be included at that point in the program.

This is extremely useful when program segments are to be linked, as global declarations (which need to be the same in each segment) can be kept in a separate file thus simplifying any alterations.

The facility cannot be nested (the included file must itself contain no £filename's).

Locate - make an executable (.COM) file which can then be run as a normal CP/M command by just typing the program name.

example:

```
locate jane
```

Creates an executable file JANE.COM from JANE.OBJ, which may then be executed by simply typing:

```
jane
```

NOTE - "locate" requires the library file PASSYS.LIB to be present on the currently logged disk.

7. Command Summary for Resident Mode.

Commands may be typed in either upper or lower case. Each line you type should be followed by <return>

E - invokes the editor (see separate documentation)

E filename - enter the editor, but first replace the program text by the contents of the specified file. The filename extension defaults to ".PAS".

R - runs an edited program (after first compiling if necessary). Starts running from the current line (usually line 0).

R xxxx - runs a program from the current line stopping at line xxxx. Source line xxxx should be executable code (not declarations). Unfortunately, due to the recursive nature of the compiler, the correspondence between line numbers and the actual source text is not always exact.

T xxxx - traces xxxx lines of source code (printing out the line numbers executed). Default one line traced.

I - initialise the stack, setting the current source line back to 0.

W - print out the current line number and the number of free bytes available.

C (or C-) - compiles an edited program. If the parameter "-" is given the program is compiled without any line numbers or variable checks in the object code. This more compact and will run slightly faster, but T(race) will not work, and no line numbers will be given with runtime errors.

L (or L-) - equivalent to C but generates a listing with line numbers on the console.

P (or P-) - as L but the listing is directed to the CP/M list device (printer).

H dddd - prints the decimal value dddd in hexadecimal form on the console.

D xxxx - prints the hex number xxxx in decimal form.

F - list the files in a diskette directory, using the syntax of the CP/M "DIR" command, for example:

```
f          - lists all files on currently logged disk
f b:       - lists all files on disk B
f *.pas    - lists all files with the extension ".PAS"
```

FW filename - writes the editor buffer to the disk file named (file extension defaults to ".PAS")

FR filename - reads the named file into the editor buffer (file extension defaults to ".PAS")

FD filename - deletes the named file from the disk.

K - clears the editor buffer. The message "Sure ?" is printed. Type "y" to confirm , anything else leaves buffer intact.

S filename - saves current OBJECT code in the named disk file.

G filename - read object code from the named file, so that it may be run without re-compiling. This command destroys any source or object text currently in memory.

NOTE - S and G only save executable code. It is a good idea to save the PASCAL source code too using the FW command, in case you want to modify the program later.

Disaster Recovery Program

The program RECOVER.COM re-enters PASCAL without clearing the text buffer, provided that the contents of the memory have not been changed by intervening commands. This is invaluable for recovering from unexpected BDOS errors or unintentional CTRL-C's !

Special Function Keys

ESCAPE - this key interrupts a program while it is running. Execution may be resumed using the R or T commands.
ESCAPE may also be used to stop a compilation.

CTRL-S - suspends execution of a program (resume by pressing any key).

CTRL-C - at any time returns you to CP/M

DELETE - deletes an input character

CTRL-U - deletes a line of input

CTRL-Z - sets EOF (input) to be true.

8. ERROR MESSAGES

RUNTIME ERRORS

1. STACK OVERFLOW - (during compilation)program is too big
   - (during execution)
program needs too much variable space or uses too many levels of recursion.

2. INTEGER READ ERROR - an integer was expected from the keyboard.

3. INTEGER OVERFLOW - overflow when multiplying two integers, or DIVing or MODing by zero, or TRUNCing or ROUNDing too large a number.

4. ARRAY INDEX ERROR - an expression used to index an array is outside the declared range.

5. VARIABLE OUT OF RANGE - a variable, or a procedure or function parameter has been given a value outside the allowed range for that data type.

6. CASE ERROR - there is no case in a case statement corresponding

to the value of the selection expression.

7. BAD PCODE - your program has been corrupted,or (hopefully not) a system bug.Occuring at random, this may indicate a memory fault.

8. SET VALUE ERROR - a set element has gone outside the range 0..127

9. FLOATING POINT OVERFLOW - may occur if the result of + - x / SQR or EXP is too large.

10. FLOATING POINT READ ERROR - a floating point constant was expected from the keyboard.

11. UNDEFINED GOTO - a GOTO statement referenced a non-existent label.

12. COMPLEX LOG OR SQUARE ROOT - attempt to take the log or square root of a negative number, or the log of zero.

13. FILE NOT OPEN FOR READING - READ or GET without a reset first.

14. FILE NOT OPEN FOR WRITING - WRITE or PUT without a rewrite first.

15. END OF FILE - attempt to read a file with EOF true.

16. FILE DOES NOT EXIST - during RESET the specified file cannot be found on the disk.

17. NO SPACE IN DIRECTORY - during REWRITE or OPEN the CP/M maximum directory size (normally 64 files) has been exceeded.

18. ERROR IN WRITING TO DISK - during PUT or WRITE. Almost always means the disk is full up.

19. ILLEGAL FILENAME - during RESET, REWRITE or OPEN.

20. FILE NOT OPEN FOR RANDOM ACCESS - attempt to do a SEEK on a file without an OPEN first.

Compiler Error Messages

1 Bad (i.e. non-Pascal) character.
2 Bad hex digit.
3 Line overflow (must be <80 characters).
4 Array index type must be a scalar (and can't be real).
5 Argument of PUT or GET must be a file
7 Bad pointer declaration.
8 'THEN' expected.

```
9 Subranges must be a scalar type other than real.
10 Type mismatch in subrange definition.
11 Type mismatch in assignment statement.
12 Function return value must be a scalar, pointer or a file.
13 Constant expected.
14 Can only apply '+' and '-' to real and integer values.
15 '..' expected.
16 Upper bound of subrange is below lower bound.
17 Identifier is not a constant.
18 'PACKED' can only be applied to a structured type.
19 Tag field type must be scalar (and can't be real).
20 Record variant label is the wrong type.
21 Procedure/function call has too few arguments.
22 Procedure/function argument does not match declaration.
23 Procedure/function call has too many arguments.
24 Type mismatch within an expression.
25 * / + - can only be applied to INTEGER or REAL data.
26 DIV and MOD can only be applied to INTEGER data.
27 Pointers may only be tested for equality.
28 < > <> <= >= = may only be applied to pointers, sets,
                                    strings and scalars
29 AND, OR may only be applied to boolean data.
30 NOT may only be applied to boolean data.
31 Not a function.
32 Error in floating point constant.
33 '(' expected after PUT or GET.
34 Illegal expression.
35 Variable expected.
36 Array index type doesn't match declaration.
37 Index on non-array.
38 Record field not found.
39 Not a record.
40 Not a file or pointer.
41 Boolean expression is required after 'IF'.
42 'WITH' statement: variable must be a record.
43 Case label is the wrong type.
44 'FOR' loop variable must be a scalar (and not REAL).
45 'FOR' loop : expression is the wrong type.
46 Identifier declared twice.
47 INTEGER constant expected after 'EXTERN'.
48 Set element has the wrong type.
49 Identifier expected.
50 ':' expected.
51 '=' expected.
52 ';' expected.
53 ')' expected.
54 'BEGIN' expected.
55 'UNTIL' expected.
56 'DO' expected.
57 ':=' expected.
58 'TO' or 'DOWNTO' expected.
```

```
59 'OF' expected.
60 Bad case label value.
61 'END' expected.
62 'NEW' argument must be a pointer variable.
63 'WRITE'/'WRITELN': field width must be an integer expression.
64 'WRITE'/'WRITELN': can only write INTEGER, REAL, CHAR,
                                    BOOLEAN or string.
65 'READ'/'READLN': can only read INTEGER, REAL or CHAR variables.
66 '(' expected after NEW.
67 PRED, SUCC and ORD can only be applied to scalars
                            other than REAL.
68 Undeclared identifier.
69 ABS and SQR can only be applied to integer and real data.
70 CHR : argument must be an integer.
71 EOLN : argument must be a textfile.
72 Illegal statement.
73 Packed array element can't be used as a name parameter.
74 '[' expected.
75 ']' expected.
76 '.' missing at end of program.
77 Missing terminator (probably one of END, ';', UNTIL or ')').
78 End of source file reached.
79 Boolean expression required after UNTIL.
80 Boolean expression required after WHILE.
81 Variable name expected after 'FOR'.
82 '(' expected after READ or WRITE.
83 '>', '<' : strict inclusion not allowed for sets.
84 Right hand side of IN must be a set.
85 Left hand side of IN must be a scalar matching base type of RHS.
86 Argument to 'PAGE' must be a textfile.
87 Base type of a set must be scalar (and can't be REAL).
88 Type incompatibility in relational expression (<>, = etc.).
89 Label must be an unsigned integer.
90 Label was not declared in a LABEL declaration.
92 Multiple label definition.
93 'FOR' variable can't be a structure member.
94 ',' expected.
```

9. Sample programs

Example 1

The character 'o' should appear to "bounce" around the VDU screen. As a variation, try deleting line 13 to produce a pattern on the screen.

```
 1
 2 program bounce (input,output);
 3 const thecowscomehome = false;
 4       DELAY = 30;
```

```
 5 var row, col, i, j, d : integer;
 6 begin
 7   row := 0;
 8   col := 0;
 9   i :=1; j:=1;
10 page;
11 repeat
12   for d := 1 to DELAY do;
13   vdu (row, col, ' ');
14   col := col+j;
15   row := row+i;
16   if (row > 15) or (row < 0) then begin
17     begin
18       i := -i;
19       row := row+i+i;
20     end;
21   if (col > 63) or (col < 0) then
22     begin
23       j := -j;
24       col := col+j+j;
25     end;
26    vdu (row, col, 'o');
27  until the cowscomehome
28 end.
29
```

Example 2 The game of Nim

```
 1     program nim;
 2     const NROWS = 16;
 3           delay = 1000;
 4           coin = "*";
 5     var pile : array [1..3] of 0..NROWS;
 6           move : record
 7                    ntaken, pileno : integer
 8           end;
 9           i : integer;
10           key : char;
11     function gameover : boolean;
12     begin gameover := (pile[1] + pile [2] + pile [3] = 0) end;
13
14     function asc (n : integer) : char;
15     begin asc := chr (n + ord ('0')) end;
16     procedure display;
17           var p, row, col, firstcol : integer;
18     begin
19           page;
20           for p := 1 to 3 do
21              begin
```

```
 22                      firstcol := p*10 + 28;
                         for row :=0 to NROWS-1 do
 24                        if pile [p] >= NROWS-row then
 25                          for col := firstcol +3 to
 26                          firstcol+5 do
 27                               vdu (row, col, COIN);
 28                        if pile [p] >= 10 then
 29                           vdu (NROWS-1, firstcol, asc (pile[p]
div 10));
 30
 31                      vdu (NROWS-1, firstcol+1, asc (pile[p] mod
10));
 32
 33               end
 34     end;
 35
 36 procedure signon;
 37         begin
 38         page;
 39         writeln ('          *** NIM ***');
 40         writeln;
 41         writeln;
 42         writeln ('I will set up three piles of coins ');
 43         writeln ('To move, take any number of coins away');
 44         writeln ('from any pile. The player who clears');
 45         writeln ('the screen wins. ');
 46         writeln;
 47         write (' Now hit any key to start : ');
 48         while getkey = chr (0) do;
 49         end;
 50
 51
 52 procedure hismove;
 53         var ok : boolean;
 54         begin
 55                 writeln ('Now enter your move :');
 56                 with move do repeat
 57                         writeln;
 58                         write ('Pile (1,2 or 3)? ');
 59                         read (pileno);
 60                         ok := pileno in [1..3];
 61                         if ok then
 62                             begin
 63                                 write ('Number to take away ?
');
 64                                 read (ntaken);
 65                                  ok  :=  ntaken  in  [1..pile
[pileno]];
 66
 67                             end;
 68                         if not ok then writeln ('What ??');
```

```
 69                  until ok;
 70                  with move do pile [pileno] := pile [pileno]
 71                                    - ntaken;
 72          end; (* of hismove *)
 73
 74   Procedure mymove;
 75   var bit : array [1..3, 1..4] of boolean;
 76         parity : array [1..4] of boolean;
 77         firstbit, x, i, j : integer;
 78         begin
 79                  for i := 1 to 3 do
 80                      begin
 81                          x := pile [i];
 82                          for j := 4 downto  1 do
 83                              begin
 84                                  bit [i, j] := odd (x);
 85                                  x := x div 2;
 86                              end;
 87                      end;
 88                  for i := 1 to 4 do parity [i] :=
 89                           bit [1,i] <> (bit [2,i] <> [3,i]);
 90                  move.pileno := 1;
 91                  move.ntaken := 0;
 92                  with move do
 93                  if not (parity [1] or parity [2] or parity
 94                       [3] or parity [4])then
 95                      begin
 96                          while pile [pileno] = 0 do pileno
 97                                        := pileno + 1;
 98                          if pile [pileno] =1 then ntaken:= 1
 99                          else
100                            ntaken   :=   random   mod   (pile
[pileno]-1)+1
101                      end
102                  else begin
103                          firstbit := 1;
104                          while not parity [firstbit] do
105                                  firstbit := firstbit + 1;
106                          while not bit [pileno, firstbit] do
107                                  pileno := pileno + 1;
108                          for i:= firstbit to 4 do
109                                  begin
110                                  x := 1;
111                                  for j := 3 downto i
do x:= x*2;
112                                  if parity [i] then
113                                  if bit [pileno, i]
then ntaken
114                                          := ntaken + x
115                                   else   ntaken   :=
ntaken - x;
```

```
116                                              end
117                                 end;
118                           with move do pile [pileno] := pile [pileno]
119                                                          - ntaken;
120          end; (* of mymove *)
121
122  begin
123   signon;
124   repeat
125          for i:= 1 to 3 do pile [i] := random mod 10 + 6;
126          display;
127          repeat
128                 hismove;
129                  if gameover then writeln ('Congratulations
...You win!')
130
131                 else begin
132                        display;
133                        mymove;
134                        for i := 1 to delay do;
135                        display;
136                        writeln ('My move was  ', move.ntaken
137                              :3,' from pile', move.pileno :2);
138                         if gameover then writeln ('***  I
win.');
139                        writeln;
140                        writeln;
141                  end;
142          until gameover;
143          write ('Another game ? ');
144          while input^ = ' ' do get (input);
145          read (key);
146          while not eoln do get (input);
147    until key = 'n';
148    page;
149    end.
150
```

PART III OXFORD PASCAL reference Manual

This manual is intended to be used for quick reference by those familiar with Pascal or a similar programming language.

1. General

1.1 Pascal keywords

These are reserved words in Pascal and cannot be redefined. They

must be written without embedded spaces or newlines. A complete list is:

and	do	function	nil	program	type
array	downto	goto	not	record	until
begin	else	if	of	repeat	var
case	end	in	or	set	while
const	file	label	packed	then	with
div	for	mod	procedure	to	

1.2 Pascal identifiers

These are names chosen by the programmer for variables, constants etc., and should consist of at least one letter, followed by zero or more letters or digits. Upper and lower case letters are equivalent. Identifiers should be unique in the first 8 characters, and must not contain embedded blanks.

The following identifiers are standard (but may be redefined):

abs	eoln	new	read	sqrt
arctan	exp	odd	readln	succ
boolean	false	ord	real	text
char	get	output	reset	true
chr	integer	pack	rewrite	trunc
cos	input	page	round	unpack
dispose	in	pred	sin	write
eof	maxint	put	sqr	writeln

(see also section 9 - extensions).

1.3 Other Special symbols

+	<	'(apostrophe)	[	:=
-	<=	.	]	;
*	>=	..	(	,
/	>	(*	)	:
=	<>	*)	^	

These symbols should not contain embedded blanks.

1.4 Comments

Pascal comments are enclosed between the composite sysbols (* and *).

Comments are totally ignored by the compiler. They can contain any characters except the corresponding closing delimiter or "*)".

1.5 Constants

Integer constants

These consist of an unsigned sequence of digits, for example

```
33  0001  0
```

No check is made to ensure that the value is less than 2**15. Integer constants must not contain embedded blanks or commas (see also section 9(a) on hex constants).

Real constants

These are of the form:

```
    <integer part> . <fractional part>
 or <integer part> E <exponent>
 or <integer part> . <fractional part> E <exponent>
```

The integer and fractional parts are non-null strings of digits. The "E" may be in upper or lower case in OXFORD PASCAL. The exponent is a digit string which may be preceded by a sign [+ or -].

Real constants must not contain ANY embedded blanks.

Examples:

```
3.14159   4E-9    -387.4E11

1E+30
```

A real constant which is out of range (greater than about 1E38) will cause an error.

Character and string constants

These are enclosed in single quotes, and may contain any character except a newline. Single quotes are included in a string by writing them twice.

Examples:

```
'c', '$', '''' (character constants)

'Hi there!', 'Fred''s string' (string constants)
```

(see also section 9(a) on hex constants).

1.6 Blanks

Any number of spaces, tabs or newlines may separate two keywords, identifiers, constants or other symbols, but at least one blank is required between adjacent keywords, identifiers and numbers.

2. Data types and operators

2.1 Integer

Pascal integers are whole numbers in the range - MAXINT to + MAXINT, where MAXINT is an implementation defined constant (32767 in OXFORD PASCAL).

OXFORD PASCAL stores integers in 16-bit 2's complement form, so integers may range from -32768 to +32767.

Integer operators are

+	addition
-	subtraction
*	multiplication
div	integer division (result is rounded towards zero)
mod	remainder operator
-	(unary operator) negation

+ and - produce 2's complement results mod 2**16.
*, div and mod are defined only on values in the range -MAXINT..MAXINT, and the result must be in this range (otherwise an error occurs).

Division by zero causes an error.

```
x mod y = x - ((x div y) * y)
```

2.2 Real

Real numbers in OXFORD PASCAL are held in floating point binary form with a 23-bit mantissa (6 1/2 digits). The exponent can range from -38 to +38.

The operators +, -, * behave as for integers, but produce a REAL result. (Which will cause an error if it is out of range).

The operator / denotes floating point division. Division by zero will cause an error.

Integer expressions and constants can be used wherever a real expression is acceptable, but real values can't be used with DIV or MOD.

Conversion from real to integer is done by the functions **TRUNC** and ROUND (section 2.8).

2.3 Char

The Pascal data type "char" operates on an ordered set of characters. In OXFORD PASCAL the 128- character ASCII set is used. (Extended to 256 characters to include the graphics characters available on some machines).

In all implementations of Pascal the digits '0' to '9' are guaranteed to be ordered and contiguous, and the letters 'A' to 'Z' are ordered (but not necessarily contiguous).

The standard functions ORD and CHR convert from character to integer and back.

```
     For example, in OXFORD PASCAL
         ord  ('A') = 65
         chr  (36)  = '$'
Also, succ(x) gives the next character after x, and
      pred(x) gives the character before x, for example
      succ('3') = '4'
      pred('1') = '0'
```

Note that in OXFORD PASCAL succ (chr (255)) and pred (chr (0)) are undefined, and chr (x) with x outside the range 0..255 is not allowed.

2.4 User-defined (enumerated) types.

These are usually defined by means of a TYPE declaration (section 3.2) for example:

```
    type day = (monday, tuesday, wednesday, thursday, friday,
saturday, sunday);

    colour   = (RED, GREEN, BLUE);
```

The data type "day" then has seven ordered values represented by the identifiers MONDAY, TUESDAY, etc.

The type "colour" has three values. The functions ORD, SUCC and PRED may be used on these types (see the previous section). For example:

```
    succ (wednesday)  = thursday
    pred (green)      = red
    ord  (monday)     = 0
    ord  (green)      = 1
```

```
ord (sunday)      = 6
```

2.5 Subrange types

The user may define subranges over any scalar type except REAL. Examples:

```
type  year = 1970..1990;
      weekday = monday..friday;
```

These types have the same properties as their parent types but often occupy less storage space. Values are checked at runtime to see that they fall in the required range. They also act as a convenient means of documentation.

2.6 Boolean

Boolean values in Pascal are represented by the standard identifiers TRUE and FALSE. In fact the data type Boolean may be thought of as resulting from the declaration:

```
type boolean = (false, true)
```

so that true>false The boolean operators defined in Pascal are:

```
and      --   logical "and" operation
or       --   logical "or" operation
not      --   (unary operator) logical negation.
```

The relational operators

```
<        --   less than
>        --   greater than
=        --   equal to
<=       --   less than or equal to
>=       --   greater than or equal to
<>       --   not equal to
```

may be used with any scalar data type (integer, real, Boolean, char, user-defined), and give a Boolean result. They may also be used to compare strings (section 5.)

2.7 Operator precedence

The relational operators

```
< > <= >= = <>  in (see section 5)
```

have lowest precedence, followed by

```
+  -  or
```

then

```
*   /   div   mod   and
```

and finally the unary operator

```
not
```

Evaluation is otherwise left to right, and can be changed by using parentheses. Particular care should be taken with expressions like:

```
(x>3) and (y=2)
```

This would be illegal if the parentheses were omitted.

2.8 Summary of arithmetic and conversion functions

FUNCTION	PARAMETER	RESULT	MEANING
abs (x)	integer	integer	absolute value
abs (x)	real	real	absolute value
sqr (x)	integer	integer	square
sqr (x)	real	real	square
sqrt (x)	real or integer	real	square root (x>=0)
ln (x)	real or integer	real	natural logarithm (x>0)
exp (x)	real or integer	real	e raised to the xth power
sin (x)	real or integer	real	sine (x in radians)
cos (x)	real or integer	real	cosine (x in radians)
arctan (x)	real or integer	real	arctangent (0 to PI radians)
trunc (x)	real	integer	convert real to integer by truncation towards zero
round (x)	real	integer	convert real to integer by rounding

chr (x)	integer	char	convert ASCII value
odd (x)	integer	Boolean	TRUE if x is odd
ord (x)	scalar *	integer	position within a data type
pred (x)	scalar *	scalar	preceding value in a data type
succ (x)	scalar *	scalar	next value in a data type

(* can't be real)

3.1 Pascal declarations and statements

Pascal Programs

A Pascal program takes the form:

```
program header
label declaration part
constant declaration part
type declaration part
variable declaration part
function and procedure declarations
BEGIN
executable statements
END.
```

The declarations are all optional. Label declarations are discussed in section 3.12, functions and procedures in section 6.

The program header is optional in OXFORD PASCAL. If it is included it consists of the keyword PROGRAM followed by a name (which can be any valid identifier) followed by a list of identifiers in brackets, for example:

```
program joe (input, output);
```

"Input" and "Output" are external files used by the program "joe" The header is terminated by a semicolon.

The final full stop after the program "end" is always required.

3.1.1 Constant declarations

These are used to assign values to identifiers which will not change throughout the program. They facilitate modifications to the

program and provide a means of documentation.

The keyword "const" if followed by one or more declarations of the form

```
identifier = value;
```

"value" may be a signed or unsigned integer, real, a Boolean, character, string, a member of an enumerated type or a previously defined constant identifier.

Examples:

```
const    message = 'hi there!';
         ch      = '$';
         PI      = 3.14;
         MINUSPI = -PI
```

3.2 Type declarations

These are used to make an identifier synonymous with a given data type. The keyword "type" is followed by one or more declarations:

```
identifier = datatype;
```

examples

```
type suit  =  (SPADES, HEARTS, DIAMONDS, CLUBS);
     int   =  integer;
     byte  =  0..255;
```

3.3 Variable declarations

In Pascal all variables must be declared explicitly. This is sometimes annoying but makes the programmer's intention clearer and helps the compiler to detect errors.

The word "var" is followed by one or more declarations:

```
identifier list: datatype;
```

Examples

```
type day = (monday, tuesday, wednesday, thursday, friday);
var x,y:real;
i :integer;
switch:Boolean;
today, tomorrow, payday:day;
favouritecolour (BLUE, RED, GREEN, PINK);
date : 1970..1990;
```

The variables denoted by these identifiers can then take any of the allowed values for the corresponding data type.

3.4 Executable statements

The executable part of a Pascal program enclosed by the keywords BEGIN and END, consists of zero or more sequentially executed statements separated by semicolons. Redundant semicolons are always accepted and generate no code. There is no need for any correspondence between the logical structure of statements and their physical layout. Well formatted programs with one statement per line are easier to read, however.

3.5 Assignment statements

The form of this statement is:

```
variable := expression
```

Where the left and right hand sides must have compatible data types. This means that they must arise from the same type identifier, or be declared as variables in the same declaration. Exceptions are if the variable type is a subrange of the expression type, or they are sets with compatible base types, or if the left hand side is real and the right hand side is integer.

The value of the variable is set to the value of the expression, and future references to the variable will yield this value.

Examples:

```
x := 3/sqrt(36)          x is set to 0.5

y := x+4;                y is set to 4.5

y := y-2                 y is set to 2.5
```

x and y are "real" variables.

3.6 Compound statements

The construction:

```
BEGIN
Sequence of statements separated by semicolons
END
```

behaves as a single statement, which when executed causes the execution of all the enclosed statements in sequence.

3.7 "If" Statements

The statement

"IF Boolean expression THEN statement 1" causes statement 1 to be executed only if the expression is TRUE. Alternatively, "IF Boolean expression THEN statement 1 ELSE statement 2" causes statement 2 to be executed instead if the expression is FALSE.

IMPORTANT - no semicolon may be placed before the ELSE.

Statement 1 and statement 2 can be any Pascal statement, including another IF statement: if x then if y then s1 else s2

is taken to mean

```
if x then
  begin
    if y then s1 else s2
end
```

3.8 "Repeat" statement

```
REPEAT
sequence of statements separated by semicolons
UNTIL Boolean expression
```

causes the sequence to be executed repeatedly (at least once) until the expression evaluates to TRUE when it is checked at the end of a loop.

3.9 "While" statement

WHILE Boolean expression DO statement 1

Statement 1 is repeated zero or more times until the expression turns out to be FALSE.

3.10 "For" statement

```
FOR variable := el TO e2 DO statement 1
```

The variable can be any scalar type except real. el and e2 are expressions of the same type as the variable. Statement 1 is executed exactly ord (e2) - ord (el) + 1 times (zero times if e2<el). On successive loops the value of the variable is el, succ(el), succ (succ(el)),..., e2

An alternative form is:

```
FOR variable := e2 DOWNTO el DO statement 1
```

Where statement 1 is executed with successively decreasing values of the variable.

Statement 1 should not try to alter the variable, as in:

```
for i := 1 to 10 do i := i + 1 (* WRONG *).
```

Structure members (section 5) can't be used as control variables in FOR loops.
Also, control variables must be local to the current block (section 6.4).

3.11 "Case" statement

```
CASE expression OF
  constant list : statement;
  constant list : statement;
      :
      :
  constant list : statement;
END
```

A redundant semicolon may be included before the END.

Each constant list consists of one or more constants (which must be the same data type as the case expression), separated by commas. The case expression must be a scalar type (and can't be real). Each label in the case statement should be unique, and indicates that the statement it prefixes is the one to be executed if the case expression has that value. If no case labels match the expression value when the case statement is executed, a CASE ERROR occurs.

WARNING - Case statements with a wide spread of values should be avoided, for example:

```
    Case n of
     1: statement 1;
     44,255: statement 2
end
```

This will generate a large jump table in memory with null entries for all the intermediate values (2,3 etc.). Generally, case statements are an efficient way of choosing one of many similar statements to execute.

3.12 "Goto" statement

Pascal statements may be prefixed by a label thus:

```
label : statement
```

The label is an unsigned integer which should differ from all other labels in the first 8 digits in OXFORD PASCAL (4 digits in standard Pascal). Control can then be transferred to this statement from another part of the program by means of the "goto" statement.

```
GOTO label
```

All labels must be declared before use (see below).

The effect of jumping into a structured statement (FOR, WHILE, REPEAT, IF, CASE, WITH) or into a function or procedure is undefined.

The use of GOTO's is not recommended if it can be avoided, since programs quickly become unreadable and error detection becomes a sticky matter.

Jumping to an undefined (as against undeclared) label is signalled as a runtime error in OXFORD PASCAL.

GOTO's can be used to exit from nested functions and procedures.

Label declaration

This takes the form

```
LABEL list of labels;
```

The labels are separated by commas.

4. Input and Output of text

A file is a Pascal structured variable which (unlike an array) has

no fixed size. Its elements are normally accessed sequentially and either reside on a disc or are associated with some physical I/O device such as the console or printer.

In this part we look mainly at textfiles, which are essentially files of characters (Pascal data type CHAR), but which give special treatment to the newline character. In particular the standard textfiles INPUT and OUTPUT, which are usually the console keyboard and display, are discussed. Disc files are covered later in parts 8 and 9 of this manual.

4.1 Outputting to textfiles

A textfile is a variable declared as type TEXT. Associated with any Pascal file f is a buffer variable f^ which is used in transferring data to and from the file. The standard procedure call:

```
write (f, ch)
```

is equivalent to

```
f^:= ch; put (f)
writeln (f)
```

sends a newline character (ASCII carriage return followed by a line feed) to the file f.

```
page (f)
```

sends a form-feed (or clears the screen in the case of the console).

4.2 Inputting from textfiles

```
get (f) reads the next item from the file f into f^.
read (f, ch)
```

for a character variable ch is equivalent to:

```
ch := f^; get (ch)
```

If the result of a GET is a newline character (a carriage return - linefeeds are ignored in textfiles), then f^ appears to contain a space and the standard function EOLN(f) is set to TRUE. Otherwise EOLN (f) is FALSE.

If the end of the file has been reached then get(f) will cause the standard function eof(f) to become true, and f^ will be undefined. Doing a get(f) while eof(f) is TRUE will cause an error.

```
readln(f)
```

skips to the start of the next line. It means:

```
begin while not eoln(f) do get (f); get (f) end
```

4.3 Reading other data types from textfiles

Syntax:

```
read (f, variable list)
```

each variable in the list can be of type CHAR, INTEGER or REAL.

char : reads one character into the variable, as above.

integer : reads any valid (signed or unsigned) integer constant into the variable, skipping leading blanks and newlines.

real : reads any valid integer or real constant into the variable, skipping leading blanks and newlines.

f^ is set in each case to the next character after the data read.

4.4 Writing other data types to textfiles

Syntax:

```
write (f, expression list).
```

each expression can be of a type CHAR, REAL, BOOLEAN, INTEGER or a string, and may be qualified by a field width

```
expression : w
```

where w is a non-negative integer expression giving the total number of characters to write to the file.

Character or string: write sufficient spaces to give a total of w characters, then write the character or string. If w is too small then the string is truncated on the right. Default w= the size of the string.

Boolean: as with string, but one of 'TRUE ' or 'FALSE' is written. Default w=6.

Integer: Write sufficient spaces first to give a total of w characters. Then write the number without leading zeros, preceded by a minus sign if it is negative. If w is too small print out the entire number with no spaces. Default w=7.

Real: There are two formats:

(i) Floating point - write a sign character (space or '-') followed by a digit, followed by a decimal point, followed by enough digits to give a total of w characters, followed by a 4-character exponent. If w is too small, at least one digit is still printed after the decimal point. Default w=12.

(ii) Fixed point - the number of decimal places must be specified:

expression : w : d

Enough spaces are first printed to give a total of w characters, followed by a minus sign if negative, followed by a decimal point and d fractional digits, with rounding if necessary. If w is too small, no spaces are printed but the entire number is still output.

4.5 Abbreviations

```
writeln (f,...) is short for write (f,...);writeln (f)

readln  (f,...) is short for read (f,...);readln (f)

write   (...) is short for write (output,...)
read    (....) is short for read (input,...)
writeln is short for writeln (output)
readln is short for readln (input)
eoln is short for eoln (input)
eof is short for eof (input)
page is short for page (output)
```

4.6 Manipulating files

There is no problem in passing files as variable parameters. In OXFORD PASCAL (but not in standard Pascal) assignment and passing as value parameters is also allowed. For example:

```
var sourcefile : text
    :
    :
begin
    sourcefile := input;
      :
      :
    read (sourcefile,x); (* reads from console *)
```

5. STRUCTURED DATA TYPES

5.1 Arrays

The syntax of array types is:

ARRAY (indextype) OF element type.

Where "indextype" can be any scalar or subrange type except real. If indextype has values ranging from m to n, say, then this defines an array of ord (n)-ord(m)+1 values of type "elementtype", which are referenced using the subscripts [m], [succ(m)],...,[n].
Alternatively arrays can be accessed as a whole:

Examples:

```
var x,z : array [1..64] of integer;

    y : array [0..3] of [-4..2] of real;
begin

    x [1]  := 0;
    x [5]  := x [1] +2;
    y [3]  [1]  := 3.345;
    z := x;          (* Transfer whole array *)
```

"element type" may be any Pascal data type. N-dimensional arrays may be abbreviated as follows:

```
array [t1,t2,...,tn] of sometype
```

which is equivalent to:

```
array [t1] of array [t2] of ... array [tn] of sometype
```

example:

```
var x: array [1..7,4..9, boolean] of char;
```

references to n-dimensional arrays may also be abbreviated,

```
x[i,7,false] := '$';
```

5.2 Sets

The syntax is:-

```
SET OF elementtype
```

where "elementtype" should be a scalar or subrange type, but not REAL. Sets are constructed from a collection of values in square brackets, for example:

```
var x : set of 0..127;
    y : set of (RED,GREEN,BLUE);
begin
    x:= [1,sqr(2), 6..74];
    y:= [BLUE, GREEN];
```

where 6..74 gives all the values between 6 and 74 inclusive. Set elements must have ordinal values between 0 and 127 inclusive. If their base types are compatible, then two sets are said to be compatible, and operations on compatible sets are:

```
*          Intersection (highest precedence)
+ and -    Union and difference
= <> <= >= Equality, inequality and inclusion tests.
```

The IN operator tests membership of a set. The right hand side should be a scalar compatible with the set's base type. IN has the same precedence as the relational operators <>,= etc.

Examples:

Assuming

```
var x,y : set of (APPLES, PEARS, ORANGES, BANANAS, FIGS);
begin
    x:=[APPLES, PEARS, BANANAS];
    y:=[BANANAS, FIGS];
```

Then

```
    x+y is [APPLES, PEARS, BANANAS, FIGS]
    x-y is [APPLES, PEARS]
    x*y is [BANANAS]
    x=y,x<=y and x>=y are all false
    x<>y is true
    y < = [APPLES, FIGS, BANANAS] is true
    y<=y is true
    y>=y is true
    y=[BANANAS, FIGS] is true
    BANANAS IN y is true
    ORANGES IN x+y is false
```

5.3 Records

The basic syntax is:

```
RECORD
    identifier list : data type;
    identifier list : data type;
    :
    identifier list : data type
END
```

An optional semicolon may be placed before the END. The fields may be accessed by the field name preceeded by a dot, for example:

```
var x,y:record
    a,b:integer;
    c:real
 end;
begin
    x.b := -33;
    x.c := 9E-20;
    x.a := x.b+2;
```

Entire records may also be assigned:

```
    x := y;
```

Several different record definitions may be combined using the following syntax:

```
RECORD
any fields common to all variants
CASE identifier:datatype OF
    constant list : (field list);
    constant list : (field list);
    :
    :
    constant list : (field list)
END
```

Again there can be a redundant semicolon before the END. The variant "field lists" may themselves contain nested variants, for example:

```
type date = record
    year : integer;
    month : (JAN,FEB,MAR,APR,MAY,JUN,JLY,AUG,SEP,OCT,NOV,DEC);
    day : 1..31;
end;
person = record
    name: packed array [1..30] of char;
    birthday : date;
    case status: (EMPLOYED,UNEMPLOYED,RETIRED,STUDENT) of
      UNEMPLOYED: (registered : date);
      EMPLOYED  : (case selfemployed : boolean of
                     true  : (numberofemployees:integer);
                     false : (employer: packed array [1..30]
                                                    of char;
                              dateemployed : date))
end;
var his:person;
begin
    his.name  := 'Harry Johnson     ';
    his.birthday.year  := 1938;
    his.birthday.month  := DEC;
    his.birthday.day  := 12;
    his.status := EMPLOYED;
    his.employer :=
          :
        etc.
```

WITH statements have the effect of declaring the fields of a record as local variables for that statement. For example:

```
with his, birthday do
    begin
        month := DEC;
        year  := 1938;
        day   := 12;
    end;
```

The record cannot however be referenced as a whole from inside the with statement.

```
WITH rl, r2, ..., rn DO statement
```

is equivalent to:

```
WITH rl DO WITH r2 DO ... WITH rn DO statement
```

5.4 Packed structures.

Records, arrays, sets and files may be preceeded by the word "packed". This is a command to the compiler to optimise storage space for that structure, possibly at the expense of speed in accessing individual components of the structure. In OXFORD PASCAL, "packed" has little effect on speed, but may cut storage by half in arrays of enumerated values, characters and subranges (0.255 and less). The disadvantage is that packed array elements can't be used as VAR parameters to procedures or functions (but whole packed arrays can).

Packed arrays [1..n] of type CHAR are special in Pascal because they are considered to be string variables of length n.

Examples:

```
var x,y :packed array [1..4] of char;
    z : packed array [1..10] of char;
begin
    x  := 'how ';
    y  := 'when';
    z  := 'Hi there !';

y=x is false, y>x, y>=x and y<>x are true.
y>'what' is true, x<'why ' is true.
```

But note:

```
x and z are incompatible (different lengths)
x and 'hello' are incompatible.
z and 'who ' are incompatible.
```

5.5 Pack and Unpack (not available in resident mode)

These standard procedures are provided in accordance with standard Pascal:

If U and P are array variables, for example

```
type t =(some data type);
var U : array [m..n] of t;
  P:packed array [a..b] of t;
where (m-n)>= (b-a)
then
   pack (U,i,P) is equivalent to
    for j:=a to b do P [j] :=U [j-a+i]
and unpack (P,U,i) is equivalent to
   for j := a to b do U [j-a+i] := P[j]
```

6. Functions and Procedures

6.1 Function and procedure definitions

The syntax for each definition is the same as the syntax for a program, except that a function or procedure header is used instead of a program header, and also a semicolon appears at the end instead of a full stop:

```
procedure or function header
label declarations
const definitions
type definitions
variable declarations
procedure and function definitions
BEGIN
executable code for this procedure or function
END;
```

Any number of procedures or functions may be defined in a program. The definitions should occur between the variable declarations and the main "BEGIN" of the program.

A procedure header has the form:

```
PROCEDURE procedurename;
```

or

```
PROCEDURE procedurename (formal parameter list);
```

A function header has the form:

```
FUNCTION functionname : datatype;
```

or

FUNCTION functionname (formal parameter list) : data type;

6.2 Procedure and function calls.

Procedure calls are statements having the form:

```
procedurename
```

or

```
procedurename  (parameters)
```

The effect is to execute any code between the BEGIN and the END of the procedure definition, and then return to continue the program normally, from the statement after the call.

Function calls are expressions which have the data type specified in the function header. To evaluate the function, any code between the BEGIN and the END of the definition is executed, and the value returned is the last value that was assigned to the function name. The value returned by a function must be a scalar or a pointer.

Examples:

```
procedure x;
begin
    writeln ('xxxxx')
end;
begin x;
    writeln ('yyyyy');
    x;
end.
```

Is equivalent to

```
begin
    writeln ('xxxxx');
    writeln ('yyyyy');
    writeln ('xxxxx');
end.
```

The following example will set i to the value 4:

```
var  i : integer;
function xyz : integer;
    begin
      xyz := 2;
      xyz := 4;
    end;
begin
    i  := xyz;
end.
```

6.3 Parameters

The usefulness of procedure and function calls can be extended by passing parameters. If these are used they must correspond in number, position and type with the formal (dummy) parameters in the definition.

The formal parameter list contains one or more parts separated by semicolons. Each part has one of the forms:

```
identifier list : datatype
VAR identifier list : datatype
FUNCTION identifier list : datatype
PROCEDURE identifier list
```

These correspond to four different classes of parameters, FUNCTION and PROCEDURE parameters which are substituted with expression values, variables, function and procedure names respectively when the function or procedure is called.

Examples:

```
const SIZE = 20;
type vec = array [1..SIZE] of integer;
var  v:vec ; i:integer;
function tan (x:real):real;
begin
    tan := sin (x)/cos(x)
end;
procedure zero (var a:vec);
begin
    for i := 1 to SIZE do a [i] := 0;
end;
function square (x:integer):integer;
begin
    square := sqr(x)
end;
function sigma (function f:integer; n,m :integer):integer;
var sum, i:integer;
begin
    sum:=0;
    for i:= n to m do sum:=sum+f(i);
    sigma:=sum;
end;
```

Given the above definitions

```
tan (0.5) would give the tangent of 0.5 radians
                              (sin (0.5)/cos(0.5))
zero (v)  would set the array v to be all zeros.
```

Note that passing large arrays (and records) as VAR parameters is a good idea, because the computer does not then have to copy the array.

```
sigma (square,1,20)
```
evaluates 1+4+9+16+...+400.

OXFORD PASCAL (and many other Pascal systems) will not let you pass standard function and procedure names as parameters, hence the need for the function "square".

WARNING - Functions and procedures passed as parameters can themselves only have value parameters, and these are not checked .

So:

```
procedure X(a:real);
begin
:
end;
procedure y(procedure b);
begin
     b(4)
end;
.
begin
y (x)
```

will lead to disaster because x expects a real and gets an integer parameter (4).

6.4 Local declarations

Any variables, constants, labels, types, procedures and functions declared within a procedure or function are local to that procedure or function and cannot be referred to from outside it.

"Global" identifiers defined outside a function or procedure may also be referenced inside it, unless they have been redefined by local definitions.

Examples:

```
program example;
var i:integer; (* may be referenced by main prog, P1 and P2 *)
    j:real;    (* may be referenced by main prog and P3 *)
    k:boolean; (* may be referenced  anywhere *)

procedure P1;
    var j:integer; (* may be refereneced by P1 and P2 only *)
    procedure P2;
    var m:char; (* may be referenced by P2 only *)
    begin
    :
    end;
begin
:
end; (* ofP1 *)

procedure P3;
const i=49;      (* may be referenced by P3 only *)
:
```

P1 and P3 may be called from anywhere.
P2 may be called from P1 or P2 only.

6.5 Recursion and forward references.

Functions and procedures can call themselves recursively:

```
function factorial (x : integer):integer;
begin
    if x=0 then factorial := 1
    else factorial := factorial (x-1)*x
end;
```

factorial (4) gives 4*3*2*1 = 24

Sometimes it is helpful for a procedure to be able to call another procedure before the procedure being called is defined.

Example:

```
procedure x(parameters for x); forward;
procedure y(parameters for y);
begin
    (* calls x *)
end;
procedure x;
begin
    (* calls y *)
end;
```

x and y call one another (they are "mutually recursive")

7. Dynamic storage and Pointers

7.1 Pointers

Variables of a pointer type take as values the memory address of other variables. This can be used in Pascal to create variables as required while the program is running, since the compiler does not need to know the memory address in advance if it can be stored in a pointer. The syntax os a pointer type is:

```
^ type pointed to
```

where "type pointed to" is an identifier which is the name of some data type (which could be declared later, allowing recursive definitions such as linked lists and trees).

Examples:

```
type  treepointer = ^tree;
      tree = record
       leftbranch, rightbranch : treepointer;
       data : sometype;
    end;
var oak : tree;
    p   : ^integer;
```

The only way of giving a pointer a value in standard Pascal is to assign it the value "nil" (which is guaranteed to point to no variable) or to use the procedure "NEW".

In OXFORD PASCAL, "nil" is the address 0000.

Pointers can, once assigned a value, be tested for equality (<> and =).

7.2 "New" and "dispose"

NEW allocates a new variable from the available storage (if any) and stores a pointer to it in the specified variable.

The variable created may then be referenced by the pointer variable followed by a^.

DISPOSE destroys the variable pointed to by the specified pointer and makes the storage available for other use. Of course you must be sure that the variable being DISPOSED is never referenced again.

Examples:

```
var p:^real;
begin
    new (p);
    p^ :=103.7;
    write (p^*p^*p^);
    dispose (p);
end.
```

Would print the cube of 103.7 and then destroy the space used to store it. P^ means the variable whose address is in p.

8. Disk Files

8.1 Declarations

Disk files are declared as Pascal variables of type "file of X" where X is the base type of the file, and can be any structured or unstructured data type. For example:

```
type patient = record
    name : packed array [1..20] of char;
    wordnumber : integer
 end;
var f: file of integer;
    g,h:file of patient;
```

very file f declared in Pascal has an asociated buffer variable f^ whose type is the base type of the file. Disk files can also be textfiles, for example:

```
var f1, f2 :text;     (see section 4.1)
```

8.2 Sequential writing

Before they can be read or written, disk files must be opened using

one of the standard procedures RESET and REWRITE.

```
rewrite (f)
```

creates an empty file which is then open for sequential writing. The end-of-file function eof(f) will return TRUE in this mode. The call put(f) writes the data in the file buffer (the variable f^) to the file.

The sequence:

```
begin f^ := expr;  put (f) end
```

may be abbreviated to:

```
write (f,expr)
```

IMPORTANT NOTE - in OXFORD PASCAL, assignments should not be made to the buffer variable f^ before a reset(f) or rewrite (f) has been done.

8.3 Sequential reading

The procedure call:

```
reset (f)
```

opens the file f for sequential reading. f must previously haave been written by a REWRITE command, otherwise the error message FILE DOES NOT EXIST will be printed. The first record in the file will be placed in the variable f^. (Or if f is empty, f^ will be undefined and eof(f) will be true).

Successive records can be read into the buffer variable f^ by the procedure call:

```
get (f)
```

read (f,x) is equivalent to x:=f^;get (f)

The function eof(f) returns TRUE when there are no more records in the file. Attempts to read past an end-of-file will cause an error.

As an example the following program writes a file containing the numbers 1 to 10, and then reads them back printing them on the console:

```
var i: integer;
    testfile : file of integer;
begin
    rewrite (testfile);
    for i := 1 to 10 do write (testfile , i);
    reset (testfile);
        while not eof (testfile) do
            begin
               read (testfile, i);
               writeln (i)
            end
end.
```

8.4 External files

The files described above are "internal"files, in other words temporary files which are normally deleted when the program (or procedure or function) in which they are defined finishes. Permanent diskette files may be created and/or accessed by giving a filename parameter to RESET or REWRITE. (The parameter may be either a string constant or a string variable). This is an extension to standard Pascal allowing specification of filenames, which can be useful in interactive programs.
If the filename is a string variable, it should be terminated by at least one space.

Examples:

```
var fname : packed array [1..15] of char;
    f, g : file of sometype;
begin
    reset (f,'datafile')
    fname := 'B:TEMP.HEX     ';
    rewrite (g,fname);
```

Disk textfile example

The following example program prompts the user for a disk file name, and then outputs an upper-and-lower-case textfile to the printer.

```
program printfile;
var  fname : packed array [1..80] of char;
     ch : char;
     f : text;
begin
     writeln;
     writeln ('Filename ? ');
     read (fname);
     reset (f, fname);
     while not eof (f) do
        begin
           while not eoln (f) do
               begin
                   read (f, ch);
                   write (printer, ch);
               end;
           readln (f);
           writeln (printer);
        end
end.
```

9. Extensions to standard Pascal

The features described in this section are specific to OXFORD PASCAL and might not be implemented on other systems.

9(a) Hexadecimal constants These are introduced by the symbol $ (for integer constants) or a backslash (for character constants)

Their main application is probably in machine language and I/O interfacing

examples:

```
const  UART=$e84f;
       linefeed= <backslash> a;
var    chardata:char;
begin
                .
                .
                .
       chardata:=linefeed; (* linefeed is a
                              constant of type CHAR *)
       poke (UART, $3f);
```

(writes the data 3f hex to a UART chip mapped at hex memory address E84F)
(<backslash> in the example above stands for the backslash symbol)

9(b) Memory, VDU and port access

The standard functions/procedures PEEK, POKE, INP, OUT, ORIGIN, GETKEY and VDU are provided for this purpose.

```
peek (x:integer):0..255
```

is a function which gives the contents of the physical memory location x, while the procedure

```
poke (x:integer; y:0..255)
```

is used to change the contents of location x to the byte y. Poke should, of course, be used with great care to avoid corrupting your program.

```
inp (x: 0..255) : 0..255
out (x,y : 0..255)
```

are like PEEK and POKE but input and output from the 8080/Z80 CPU port x.

```
origin (x: ^sometype, y:integer)
```

sets the pointer x to point at the physical memory location y. x can be any pointer type. This should be used with care (see section 10)

The procedure VDU (x,y :integer; c :char) stores the character c in the VDU memory row x, column y.

WARNING----"VDU" takes no account of scrolling. To ensure that row 0 column 0 is in the top left corner of the screen, the screen should be cleared (using the standard procedure PAGE) before VDU is used.

Finally, the function

```
getkey:char
```

returns a character read directly from the console keyboard port. Chr(0) is returned if no character is ready.

EXAMPLES

```
var x:0..255;
```

```
begin poke($014c, $33);     stores the byte 33h at address 41Ch
x:= peek(47);               sets x to the contents of decimal
                            memory address 47

write (inp ($C9))           reads a value from the CPU port
                            C9 and prints it

out ($12, ord ('$'))        writes a dollar to the
                            CPU port 12h

page;                       clears the VDU screen

VDU(0,3,'?');               writes a question mark to the VDU
                            row 0, column 3

while getkey=chr(0)do;      waits for someone to press a key
```

9(c) Hexadecimal input and output

The procedures WRHEX and WRHEX2, and the function RDHEX are provided.

```
wrhex (f:text; x:integer)
```

writes x as four hex. digits on the textfile f.

```
wrhex2 (f:text; x:0..255)
```

writes the byte x as two hex. digits.

Examples :

```
wrhex (printer, -1); wrhex2 (output, 3)
```

prints FFFF on the printer and 03 on the console.

The function

```
rdhex  (f:text):integer
```

reads a 16 bit value from the file f, skipping any leading blanks and discarding all but the last four digits read.

9 (d) Bit manipulation

ANDB,ORB,XORB,NOTB,SHL, and SHR are functions operating on integers but treated as 16 bit logical data. The first four do bitwise AND, inclusive OR, exclusive OR and 1's complement.

SHR(x,y) shifts x left by y bits(zeros are shifted in)

SHL(x,y) shifts x right by y bits

SHL(x,-y) is equivalent to SHR(x,y)

examples:

```
andb ($fff0,$00ff)=$00f0

orb ($ff00,$000f) = $ff0f

xorb($ff00,$0ff0) = $f0f0

notb($f0f0) = $0f0f

shl (4,4) = $40

shl(4,-1) = 2

shr (4,-12) = $4000

shl(4,0) = shr(4,0)=4

shr ($4444,4) = $444
```

9(e) Catching I/O errors

Occasionally it is necessary for a program to protect itself against unexpected termination due to invalid input.
The procedure call

```
iotrap(false)
```

turns off Pascal error messages for real and integer read operations and disk I/O

```
iotrap(true)
```

turns checking back on again. After each integer or floating point or hex read operation the function IOERROR may be used giving an integer error number:

```
ioerror= 0-No error
         2-Integer read error
         10-Floating point read error
         etc. (see section II.8 for a complete list of I/O runtime errors).
```

9(f) Keyboard interrupts

The calls

```
breaks(true)
breaks(false)
```

enable and disable the ESCAPE (and CTRL-S) keys respectively.

The default is breaks (true).

9(g) Random Number Generator

The function random :0..255 gives a random no. between 0 and 255. A pseudo-random generating sequence is used but this is initialised by timing all keyboard inputs and is also "kicked" frequently by the Pascal interpreter.

The construction

```
random+(random mod 128)*256 generates a random no.
                            between 0 and MAXINT, while

random mod n+1              generates an (almost)
                            random no. in the range
                            1..n if n is not too
                            large.
```

9(h) Underscore

The character ' ' is allowed as a letter in identifiers giving improved readability.

9(i) Input of String Variables

String variables (ie packed arrays [1..n] of char) may be read from textfiles in a similar manner to characters, integers and reals. Any leading spaces or newlines are first skipped, then an entire line of characters is read from the file into the string variable. If the string is too long, it is truncated on the right, if it is too short it is padded out with spaces.

A major application is for inputting file names from the console.

9(j) Program chaining (disk mode only)

The OXFORD PASCAL command :

```
     chain (filename)
```

stops execution of the current program and invokes the program named. The value of GLOBAL variables will be preserved only if declarations are identical in the old and new programs. All files are closed.
The filename can be either a string or a string variable. (If a string variable, at least one space must be used as terminator).
When used under the RUN command, a ".obj" extension is impiled.
When used in a LOCATED program, the chain command simply executes

the CP/M command named (.COM extension assumed), which need not be a Pascal program.

Example:

```
file "progl"  (object code in "progl.obj"):
    begin
        writeln ('First program');
        chain ('Prog2')
    end.

file "prog2"  (object code in "prog2.obj"):

    begin
        writeln ('Second program');
        chain ('Progl')
    end.
```

The command
```
ex progl
```
would cause the following to be printed:
```
First program
Second program
First program
Second program
.
.
```
until the ESCAPE key is pressed.

Program chaining is a useful technique for splitting up large programs, or for menu-driven applications.

9(k) CP/M Directory maintainence (disk mode only)

Deleting and renaming files.

delete (fname)	deletes the file "fname" from the CP/M directory.
rename (fname1,fname2)	renames the file "fname1" to be "fname2"

examples:

```
var oldfile : packed array [1..6] of char;
begin
    delete ('b:test');
```

```
delete ('mary.com')
oldfile := 'a:joe ';
rename (oldfile, 'smith');
```

The currently logged disk.

```
procedure login (drive : char)
```

logs in a new disk, for example:

```
login ('B');
```

To find out the drive letter of the currently logged disk, use the function:

```
logged : char
```

9(1) Random Access Files (disk mode only)

A file may be opened for random reading or writing by using the OPEN command (which has the same syntax as RESET and REWRITE). The file is automatically positioned at record zero (the first record) and this record if it exists, is placed in the file buffer.

```
seek (f, n)
```

positions the file f at the nth record, leaving the nth record in the file buffer. A subsequent PUT will write the nth record and advance to the n+1th, while a GET will advance to the n+1th record and read it into the file buffer.

Eof (f) has no significance when in random access mode. Sequentially written files may always be read randomly, but randomly written files are likely to have unwritten (garbage) records which may contain an end of file, and should not be read sequentially.

Random Access Example

```
(* A simple mail list program using random access files.
 *
 * The file MAIL.LST must first be formatted (see separate
 * program below)
 *)

const MAXREC = 100;  (* Arbitrary max. number of records *)

type data = record
        name : packed array [1..30] of char;
        address : packed array [1..60] of char;
```

```
        telephone : packed array [1..15] of char;
    end;

var  fyle : file of data;
     rec : data;
     command, cmd : packed array [1..10] of char;
     recnum, i : integer;

procedure getrecordnumber;
     begin
        repeat
           writeln;
           write ('Record number ? ');
           read (recnum)
        until (recnum > 0) and (recnum <= MAXREC)
     end;

procedure getname;
     begin
        write ('Name ? ');
        read (rec.name)
     end;

procedure getaddress;
     begin
        write ('Address ? ');
        read (rec.address)
     end;

procedure gettelephone;
     begin
        write ('Telephone number ? ');
        read (rec.telephone)
     end;

begin
     open (fyle, 'mail.lst');
     repeat
          page;
          writeln ('Type:  i to input new record');
          writeln ('       c to change an existing record');
          writeln ('       d to display a record');
          writeln ('       l to list contents of the file');,
          writeln ('       q to quit');
          writeln;
          write ('? ');
          read (command);
          if command [1] in ['i','c','d','l'] then
            case command [1] of

              'i': begin   (* insert new record *)
```

```
                    getrecordnumber;
                    getname;
                    getaddress;
                    gettelephone;
                    seek (fyle, recnum);
                    write (fyle, rec);
                end;

            'd': begin   (* display a record *)
                    getrecordnumber;
                    seek (fyle, recnum);
                    writeln;
                    writeln ('Name:      ',fyle^.name);
                    writeln ('Address:   ',fyle^.address);
                    writeln ('Tel:       ',fyle^.telephone);
                end;

            'c': begin   (* modify a record *)
                    getrecordnumber;
                    seek (fyle, recnum);
                    rec := fyle^;
                    writeln;
                    write ('Which field do');
                    write (' you wish to change ? ');
                    read (cmd);
                    if cmd [1] = 'a' then getaddress
                    else if cmd [1] = 'n' then getname
                    else if cmd [1] = 't' then gettelephone;
                    write (fyle, rec);
                                                    end;

            'l': begin  (* list the file *)
                    seek (fyle,1);
                    for i:=1 to MAXREC do begin
                        read (fyle, rec);
                        writeln;
                        writeln (i);
                        writeln (rec.name);
                        writeln (rec.address);
                        writeln (rec.telephone);
                    end;
                end;
        end;  (* of case statement *)
    until command [1] = 'q';
end.


(* Formatting program to create an empty mail list file *)

const MAXREC =100;
type  data = record
```

```
      name : packed array [1..30] of char;
      address : packed array [1..60] of char;
      telephone : packed array [1..15] of char;
    end;

var   fyle : file of data;
      rec : data;
      i : integer;

begin
      rewrite (fyle, 'mail.lst');
      name := '                              ';
      address :=
        '                                                    ';
      telephone := '               ';
      for i := 0 to MAXREC do write (fyle, rec)
end.
```

10. OXFORD PASCAL interface guide

The purpose of this section is to provide all the necessary information to write 8080 (or Z80) machine language subroutines for OXFORD PASCAL programs.

10.1 Assembly language format

Assembly language routines are declared as Pascal functions or procedures but the body is replaced by the word "extern" followed by an integer constant (the routine address). Any parameters are passed on the stack and should be removed by the assembly language routine. The routine should also push a return value on the stack if it is declared as a function. The best way to describe this is by example, so here is a simple function to add two integers:

```
program test;
function addxy (x,y: integer):integer;
    extern $7400;
begin
    write (addxy(3,4));
end.
```

This should result in the output:

```
7
```

Provided that the assembly language routine is correctly located at memory address 7400h:

```
        org   7400h
addxy:  pop   b         ; pop return address
        pop   h         ; pop the parameter y
        pop   d         ; pop the parameter x
        dad   d         ; add x to y
        push  h         ; put the result on stack
        push  b         ; put the return address back
        ret             ; return to Pascal
```

Where to locate assembly langauge routines

In executable (.COM) files produced by the "locate" command, locations 2D00 and 2D01h point to the last location used by the Pascal object code. Locate the routines somewhere above this address and modify the pointer accordingly. (The SID and SAVE commands can be used for this purpose). Remember that the first byte of a .COM file actually corresponds to location 0100h.

10.2 Storage formats

All scalar and subrange types (except REAL), and pointers are passed as 16-bit words in the usual low-high format.

Reals are passed as 4 bytes:

```
loc n+3: 8 bit 2's complement exponent
loc n+2: Sign, MS 7 bits mantissa.
loc n+1: Middle 8 bits of manissa.
loc n  : LS 8 bits of mantissa.
```

If the exponent is zero then the mantissa MS bit represents 0.5. Arrays are stored row-by-row (the opposite to FORTRAN), the lowest element has the lowest address.

Arrays are byte-packed if their elements are scalars in the range 0..255 (eg. char), and "packed" was specified. In this case the size is always rounded up to an even number of bytes.

Records are stored with their fields in reverse order (first declared has highest address). Sets are passed as a 128-bit map, a "one" indicates membership. Odd and even bytes are reversed:

```
loc.  n+15:     bit 15 .... bit 8
loc   n+14      bit 7  ...  bit 0
  :
loc   n+1       bit 127 ... bit 120
loc   n         bit 119 ... bit 112
```

IMPORTANT - pointers always point to the location above the highest

byte used by the actual data. This also applies to VAR parameters,which are passed as addresses.

Example:

```
const  VDUSIZE = 1024;   (* 16 rows of 64 chars *)
type screen = packed array [1..VDUSIZE] of char;
var  vduptr :^screen;
begin
    origin (vduptr, $9000 + VDUSIZE)
    :
    :
```

This declares an array based on the Triton vdu address 9000h. vduptr^ [1] is the first vdu location.

CP/M File Format

Data is stored on disk in an identical manner to the way packed arrays are stored in memory.

The end of file is marked by a control-Z (1Ah) followed by 0 to 127 zero bytes to pad out the last record. Random files have no end of file. Text files need only have a CTRL-Z to mark the end of file (or not even this if the end of file coincides with the end of a record). This ensures compatibility with CP/M editor files.

V. A tutorial introduction to the OXFORD PASCAL text editor

Introduction

Our discussion of the Text editor will assume that it is being used in conjunction with the OXFORD RESIDENT PASCAL compiler. Once you have entered the editor program the actual configuration in use is irrelevant.
This manual is intended to simplify learning the use of the editor and it is recommended that the examples and exercises are followed. Do the exercises! They cover material which is not completely covered in the text. An appendix covers all of the commands available.

Clearing the buffer the KILL command K

To clear the text buffer, either because it contains rubbish, or because you do not want the contents, use the KILL command. This is entered simply as the letter K.
Do not forget the RETURN afterwards. The editor will ask :

```
Sure ?
```

You need to type 'Y' RETURN to confirm that you want to kill the buffer. We now have an empty buffer ready to put our text into. After entering this command, the editor will reply with its prompt (a >) to indicate that it is waiting gor your next command.

Creating text the APPEND command A

Initially we will just deal with the case of entering text into the empty buffer and adding on lines to the end of those already entered. The append command is written simply as the letter

```
a
```

It means "append lines of text to those already in the buffer as I type them in". Appending is rather like adding new sentences to an essay. To enter text, we type a RETURN and then the lines of text that we want, like this:

```
Now is the time for
all good men to come
to the aid of their party.
```

Notice that the last line is a single period. This is not a line of text and it is not put in the text buffer. The "." signifies to the editor that we want to stop adding new text to the buffer. If later on you find that the editor is ignoring your commands, type a line containing a single period. Even experienced users forget to type that line sometimes. If this happens, you will probably find a few lines of rubbish in the text which will need deleting later. After the above append command has been completed the text buffer contains the three lines of text but not the "a" or "." which were commands. To add more text, simply give another "a" command and continue typing.

Errors - '?'

If at any time you make an error in the commands that you give, the editor will respond with a ? . This is about as cryptic an error message as you can get, but you will soon realise why the editor cannot make sense of your command. The two most common errors are specifying a line number which does exist or giving a command which does not exist.

Printing the contents of the text buffer

Having entered some text into the text buffer we will want to print it so that we can check it for errors. To do this we use the print

command:

```
p
```

This is done as follows. We specify the lines we want to print before the p as a start and end line number separated by a comma. At present we have 3 lines in the text buffer so 1,3p will print all of the lines in the text buffer. If we only want to print one line, say the second, we can omit the second number and in this case simply type 2p rather than 2,2p. In our case this would produce

```
all good men to come
```

If we want to print all of the lines in the buffer, then the editor allows us to use another abbreviation which is especially useful when there are a lot of lines in the text buffer and we do not know exactly how many. We can use the symbol $ as the second line number so 1,$p means print all of the lines in the buffer from the first to the last.

In general, when a line number is required in a command we can use $ to represent the last line number. To stop a listing on the screen type an escape. This will cause ? to be printed followed by a prompt.

If we wish to print single lines of the buffer it is possible to abbreviate the command even more than shown above. It is simply necessary to type the line number and omit the p.

For instance, 2 will print line 2 or $ will print the last line. Finally we can modify $ by typing something like $-1 meaning the last but one line. Hence $-1,$p will print the last two lines in the text buffer. To find how many lines are in the buffer type '$='.

Leaving the editor the quit command 'q'

To leave the editor and do something with the text we have just put in the buffer, type q. After this you are returned to Pascal. The text buffer is left intact and can be used as data for an assembler, a compiler or a text processor.

Exercise 1

Enter the editor from Pascal and clear the text buffer. Then enter some text using append. Use a series of commands like:

```
k
a
....text....
.
```

Now experiment with the print command to list all or parts of the buffer until you are confident how print works. An understanding of the use of line numbers is important because the same format is used for other commands later on. Also verify that if you quit and re-enter Pascal then the text remains unchanged until a kill command is issued. If you try to print and empty buffer or line 0, or a line past the end of the buffer or part of the buffer backwards (i.e. 3,1p) ...then an error will occur.

Writing text out to a file the write command 'w'

Having put text into the buffer and modified it, you will probably want to save it on tape for later use. The write command allows you to save the whole of the text buffer on disk as a named CP/M file. Type a 'w' followed by the file name, for example:

```
w program
```

The file extension defaults to .PAS, so that the buffer will be written to the file PROGRAM.PAS. If the file exists already it is first deleted.

Reading text back from a file the read command 'r'

When you wish to reload the text you saved on disk, use the r command. Again you need to give a file name. This should be exactly the same as the name used to save the file.

Exercise 2

Using the commands described so far, put a few short pieces of text onto disk files and use the read command to reload these.

The current line 'dot' or '.'

Suppose that the text buffer contains the three lines we had above and you have just typed 1,2p. The editor has just printed the first two lines. Try typing just

```
p  (no line numbers)
```

The result is that

```
all good men to come
```

is printed, which is our second line. More significantly, it is the last line that we did anything to (we printed it). We can repeat

this p command and line 2 will continue to be printed out. The editor maintains a record of the last line that you did anything to and allows you to use it as an implicit line number, rather than specifying one explicitly. This line is referred to by the shorthand notation

. (pronounced "dot")

Dot is a line number just like $ and it is referred to as the current line. We can use it in many ways, for example:

.,$p

This will print all of the lines in the text buffer from the current line to the last (in this case lines 2 and 3).

Most commands affect the value of dot. The print command sets dot to the last line printed. In the above case we arrive at .=$=3.

Dot is most useful when it is used in combinations like

.+1p

which will print the next line. Remember that the p can be omitted so this command is equivalent to .+1. This gives us a useful way to list through the text buffer. We could also use:

.-1

to print (and move back to) the previous line in the buffer. Another useful command which will print the previous three lines of the text buffer is:

.-3,.p

Remember that all of these commands change dot to the last line printed. Before looking at any new commands we will summarise a few things about dot and the print command. P can be preceded by 0, 1 or 2 line numbers. If no number is given, then the current line is printed. If one number is printed then that line is printed and dot is moved to that line. If two numbers are given then all the lines between the two numbers (inclusively) are printed and dot is moved to the last line printed. Also if two lines are specified then the first must be less than or equal to the second (see Exercise 1).

Finally the editor allows us to use a further degree of shorthand to move back or forward by one line. Typing RETURN will cause the next line to be printed - it's equivalent to .+1p. Try typing ^ followed by return; it is the same as .-1p.
To find out where you are in the buffer type '.=' and the value of dot is printed.

Deleting lines the 'd' command

If we want to get rid of some of the lines in the text buffer we use the 'd' command. Just like the p command it can be preceeded by 0, 1 or 2 line numbers. It behaves in the same way except that it deletes lines rather than printing them. Hence:

```
2,$d
```

will delete all the lines from line 2 to the end of the buffer. We can check this by typing 1,$p. Notice that $=1 now.
Dot is set to the line after the last line deleted unless this was the last line in which case dot is set to the new last line.

Exercise 3

Experiment with the p, a, r, w and d commands until you understand fully how they work and how dot, $, and line numbers are used.

Notice that the r command sets dot to line 1. If you are adventurous, try putting a line number before the 'a' command (e.g. 3a). You will find that text is appended after the line number specified rather than after the current line. Dot is set to the last line of text which is added.

Modifying text the subsitute command 's'

We now come to one of the most important commands available which allows us to change text strings. The command is used to change single letters or groups of letters. An example of its use is for correcting spelling or typing mistakes. Suppose that line one said "Now is th time for". We can insert the 'e' into 'the' with the following command:

```
1s/th/the/
```

This command means "in line one substitute 'the' for the first occurrence of 'th'." To verify that we have achieved the correct result, type 'p' and the line will be printed. Notice that the substitute command must move dot to that line. The general form of the substitute command is

```
starting line, ending line s/string 1/string 2/
```

and string 2 is substituted for string 1 for the first occurrence of string 1 on each line between the starting line and ending line. The rule for line numbers is exactly the same as for the print command,

and dot is set to the last line altered. This is a trap for the unwary because if no substitution occurred then dot is not moved. We can use

```
l,$s/speling/spelling/
```

to correct the first spelling mistake in the word 'spelling' on each line. This is useful for people who are consistent mis-spellers! If no line number is used then the substitution occurs on line dot, e.g.

```
s/something/something else/p
```

This makes a correction on the current line and prints it (a 'p' can optionally follow at the end of the substitute command). It is also possible to say:

```
s/something//
```

which will delete 'something' from the current line.

Exercise 4

Experiment with the substitute command. In particular use it on a string which occurs sevaral times on a line. Try this:

```
a

the other side of the coin

s/the/on the/
```

You will get:

```
on the other side of the coin
```

A substitute command changes only the first occurrence of a word on a line. You can change all occurrences of a word by adding a 'g' (meaning global) to the s command like this:

```
s/-----/------/gp
```

If you want to change a word or phrase with a '/' in it, you must precede the slash with a backslash, e.g.

```
s/5 /3/5+3/
```

will change 5/3 to 5+3.
Also to change a in the text you must put in the string, e.g.

```
s/  //backslash slash/
```

will change / to the words backlash slash.

Context searching - /----/

Now that we have mastered the substitute command we can move on to discover another way of specifying line numbers. Suppose we have our original three lines of text in the buffer:

```
Now is the time for
all good men to come
to the aid of their party.
```

and we wanted to find the line containing 'their' and change it to 'the'. Of course, with only three lines of text in the buffer it is fairly easy to do this, but with a large buffer full of text things would not be so easy; context searching is simply a good method of specifying a line without regard for its number by giving some context on it. The format used is:

```
/string of charaters we want to find/
```

More specifically for the example mentioned above

```
/their/
```

is sufficient to find the desired line. Having found the required line, the editor moves dot to that line and prints it for verification, giving:

```
to the aid of their party
```

In doing the search , the editor looks for the next occurrence of the specified string. This means that is starts searching from line .+1 to line $ and then it wraps around and searches from line 1 to . If the required match does not occur, then the usual error message is printed (?) and dot is unchanged. We can use a context search anywhere that a line number is required so we can acheive the required alteration by using

```
/their/s/their/the/p
```

This command has three parts, a context search, the substitute command and the print command.

A context search can also take the form of a simple expression so in our example text, the following are all references to line 2.

```
/Now/+1,  /good/,  /party/-1,  $-1,  2
```

The choice of how to specify a line is dictated purely by convenience, for example:

```
/Now/,/Now/+2p
```

will print 3 lines. (In fact it prints the whole buffer.)

Exercise 5

Experiment with context searching. Try a body of text with several occurrences of the same string of characters and scan through it using the same context search. Notice that if the text string occurs twice on the same line then successive searches find different lines. Use context searches to specify line numbers for commands other than substitute. Beware of finding a string on a different line from the one you expect!

Change and Insert - c and i

The change command is used to replace one or more lines with any number of new lines and the insert command is used for inserting a group of one or more lines. For instance to change lines '.+1' through '$' to something else, type

```
.+1,$c
--- type the new lines of text here ---
```

The text typed between the change command and the . will take the place of the original lines. Dot is set to the last new line entered. The single '.' at the end of the command shown is used in the same way as the '.' in the append command to leave the text entry mode.

Insert is similar to append in that it puts fresh lines into the text buffer but it puts the text before the line specified. Dot is set to the last line inserted.

Exercise 6

Change is rather like a combination of delete followed by insert. Experiment to verify that

```
start,end d
i
----text----
.
```

is like

```
start,end c
----text----
.
```

Experiment with a and i to see that they are similar, but not the same. You will observe that

```
line number a
----text----
.
```

appends after the given line number, while

```
line number i
----text----
.
```

appends before it. Notice that if no line number is given , a appends after line dot , while i inserts before line dot.

Moving text around the 'm' command

The final command allows us to move groups of lines around in the text buffer. For instance, the order of procedures can be easily changed in a Pascal program, especially if we use context searches to specify the line numbers. Unlike the commands we have used so far, the move command requires three line numbers. The format of the command is:

start line, end line m after this line

It moves all of the lines between the start and end lines so that they follow the third line number, e.g. 1,3m$ will move the first three lines of text to the end of the buffer. Dot is set to the last line to be moved. The third line number must be exclusively outside the range of the start and end lines. If the text buffer contained :

```
First paragraph
----
end of first paragraph

Second paragraph
----
end of second paragraph
```

then we could reverse the two paragraphs by the following command:

```
/Second/,/second/m/First/-1
```

Notice the '-1'. This is because the moved text goes after the lines specified.

Summary of commands and line numbers

The general form of editor commands is the command name, perhaps preceded by one or two line numbers. Only one command is allowed per line, but a p command may follow the search command.

a (append)

Add lines to the buffer (at line dot, unless a different line is specified). Appending continues until '.' is typed on a new line. Dot is set to the last line appended.

c (change)

Change the specified lines to the new text which follows. The new lines are terminated by a '.'. If no lines are specified, replace line dot. Dot is set to the last line changed.

d (delete)

Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless $ is deleted, in which case dot is set to $.

i (insert)

Insert new lines before specified line (or dot) until a '.' is typed on a new line. Dot is set to the last line inserted.

m (move)

Moves lines specified to after the line named after m. Dot is set to the last line moved.

p (print)

Print specified lines. If none specified, print line dot. A single line number is equivalent to 'line-number p'. A single return prints '.+1', the next line. An ^ followed by return is equivalent to '.-1p'

q (quit)

Exit from the editor.

r (read)

Read a named file into the buffer. Dot is set to the first line read.

s (substitute)

s/string 1/string 2/ will substitute the characters of 'string2' for 'string 1' in specified lines. If no line is specified, make substitution in line dot. Dot is set to the last line in which a substitution took place, which means that if no substitution took place, dot is not changed. S changes only the first occurrence of string 1 on a line; to change all of them, type a 'g' after the final slash.

w (write)

Write out buffer to a named disk file. Dot is not changed.

.= (dot value)

print value of dot. ($= prints value of '$'.)

/----/ (context search)

Search for next line which contains this string of characters. Print it. Dot is set to line where string found. Search starts at '.+1', wraps around from '$' to 1, and continues to dot, if necessary.

OXFORD PASCAL

AMSTRAD CPC 6128 PCW 8256

WHY PASCAL?

Since Jensen & Wirth specified the language in 1975, PASCAL has become the world's most popular programming language. It is used in nearly all universities and recommended by educational authorities in the UK, Europe and the USA.

A programmer who learns PASCAL can easily master other structured languages such as MODULA 2, C, PLI, BCPL, and ALGOL.

A REAL PASCAL COMPILER FOR YOUR MICRO.

OXFORD PASCAL is an extended full implementation of this highly acclaimed programming language. In PASCAL you can write programs which could never be implemented in BASIC or assembler.

OXFORD PASCAL includes:

- Fully recursive procedures and functions
- Record structures
- Sets
- Type definitions
- WHILE, FOR, REPEAT-UNTIL loops
- CASE statements
- Files
- Full linker

in fact, everything in Jensen & Wirth's original definition of the language. In addition, OXFORD PASCAL provides many extensions.

EXTRA

OXFORD PASCAL features a whole range of extensions designed to make maximum use of your Amstrad. These include Hex constants and I/O, Bit manipulation, Random access files, Random numbers, Program chaining, Separate compilation, Linking and more.

OXFORD PASCAL IS FAST

OXFORD PASCAL compiles down to FAST COMPACT P-code, giving you the real speed and power of Pascal, together with the ability to compile very large programs.

OXFORD PASCAL IS COMPACT

Because it compiles into P-code, OXFORD PASCAL reduces programs into the most compact form possible. In fact it allows you to pack more code into your 64 than any other language, and should your programs become too large, you can still use the CHAIN command to overlay limitless additional programs without losing data.

DISC AND RESIDENT COMPILER

Under the resident compiler, programs can be written and run on the spot without disc access. Compilation is fast enough to make using the system much like using the BASIC interpreter. A TRACE option allows you to follow execution of a program with direct reference to the source code.

Under the Disc compiler very large programs can be developed which utilise the whole of memory for Pascal object code. The LINKER allows complex programming tasks to be broken up into easily manageable, separately compilable files.

®CP/M is a registered trademark of Digital Research.

Oxford Computer Systems (Software) Ltd.
Hensington Road, Woodstock, Oxford OX7 1JR, England.
Telephone (0993) 812700 Telex 83147 Ref.OCSL